SIZE BOUNDS AND PARALLEL ALGORITHMS FOR NETWORKS

Gavriela Lev nee Freund

Ph. D.

University of Edinburgh

1980

I declare that this thesis has been composed by myself.

I declare that this work is my own except where otherwise indicated.

June 1980

ABSTRACT

In this thesis we discuss the size bounds of, the construction of, and routing algorithms for certain types of connection networks.

For superconcentrators we give a lower bound of $5N-o(N)$ edges, and when the indegree is restricted to 2 of $4N-o(N)$ nodes. This implies a lower bound of $4N-o(N)$ on the number of additions in Discrete Fourier Transforms of prime order. We improve the non-constructive upper bound from $39N+o(N)$ edges (or $40N$ edges) to $38.5N+o(N)$ edges (or $39.05N$ edges) and from $36N+o(N)$ computed nodes to $32.5N+o(N)$ computed nodes. We give an explicit recursive construction with less than $3N\log_2 N$ edges, which is better by 20% than the construction of permutation networks, which up till now have served as the best explicitly constructed superconcentrators for reasonable input sizes. For hyperconcentrators we give a construction of size $1.5N\log_2 N+o(N\log N)$. This result gives an improvement on the size of explicit concentrators and generalized connection networks by small multiplicative factors.

Routing in networks constructed recursively can be performed easily when the the input size of the components, d, is a power of 2. We give a parallel algorithm for routing in permutation networks in time $O(\log^2 N)^{(*)}$ when d is a power of 2. We give a parallel routing algorithm in $O((d/\log_2 d)\log_2^3 N)$ time when d is arbitrary, and present the improved algorithm suggested by Pippenger, which requires $O(\log_2^3 N)^{(*)}$ time. Both algorithms use as a subroutine a parallel edge colouring algorithm. The main vehicle in producing the parallel algorithm is a "parity labelling" algorithm on "oriented" lists. Another application of the parity labelling algorithm is a parallel maximal (not maximum) matching algorithm of time complexity $O(\log^4 N)$. The exact model of computation used in those algorithms is also presented.

(*) independent of d.

ACKNOWLEDGMENTS

I wish to thank warmly my supervisor Leslie G. Valiant for his encouragement and valuable advice.

Many thanks are due to my husband Yaacov Lev for his patience during this period.

I am also grateful to Dana Angluin for many useful discussions at the early stages of this work.

Finally I wish to acknowledge the support given to me by the Computer Science department of the University of Edinburgh and the Edinburgh Postgraduate Studentship provided by the university.

CONTENTS

To the Memory of my Father

Robert Freund

1 INTRODUCTION

1.1 Introduction

Two main interpretations of networks are discussed in the following pages:

a) Networks as switching circuits.

b) Networks as program execution graphs.

When networks serve as switching circuits their aim is to supply specified interconnections between various points in a system. When they serve as program execution graphs, they describe the interdependence of computation steps in algorithms. In investigating the properties of networks we use terms borrowed from both fields.

Minimizing the size of a network means minimizing the number of switches in a switching circuit or minimizing the number of computation steps when the network represents a computation. This is equivalent to minimizing the number of edges or of nodes respectively. When parallel operation is considered, i.e. simultaneous message passing, or parallel algorithms, the depth of the network corresponds to time delay in both interpretations.

In the sequel we give constructions for certain related networks: superconcentrators, hyperconcentrators, etc., of size $O(N \log N)$ and depth $O(\log N)$. In these two networks the constructions require $3N\log_2 N+o(N\log N)$ and $(3/2)N\log_2 N+o(N\log N)$ edges respectively, and are better than the known explicit constructions for all reasonable size networks. For superconcentrators we give a lower bound and an improvement to the nonconstructive upper bound of $5N-o(N)$ and $38.5N+o(N)$ edges respectively. When counting nodes we get the bounds $4N-o(N)$ and $32.5N+o(N)$ respectively when the indegree is restricted to two.

Superconcentrators and hyperconcentrators find use as switching networks and as mathematical computation graphs.

Routing is an important problem in switching networks. We present parallel routing algorithms for permutation networks and superconcentrators. Routing in the other networks is easily derived from their construction. If the network for which the routing is sought is built from subnetworks whose input size is a power of two, the parallel routing algorithm requires $O_d(\log^2 N)$ (*) time. Otherwise it requires $O((d/\log_2 d)\cdot\log_2^3 N)$ time. We also give the improved algorithm suggested by Pippenger, which requires $O_d(\log^3 N)$ time. The model of computation used in those algorithms is a powerful model which allows simultaneous accesses to a global memory by every processor.

1.2 Switching circuits

The interest in connection networks in the context of computers is three fold:

a) Distributed processing is becoming a common feature in many big organizations. For example computerized selling points which update stock, airline reservation systems, and the big national systems such as Arpanet, Telenet, EX2 and Euronet.

b) Parallel processing becomes more attractive with new technological developments (e.g. VLSI). Here a system of several processors, each with its own memory, is engaged in solving one problem. The system requires interconnections among the processors and possibly between the processors and some global memory. The interconnection pattern determines (i.e. restricts) the efficiency of computation in this case. (See Siegel (1977,1979), Lawrie (1975), Stone (1971)). The VLSI technology prompts research into connection networks complying with the restrictions of the technology. (Guibas, Kung and Thompson (1979),

*O_d indicates that the multiplier in the bound is independent of d

Preparata and Vullemin(1979), and see also Kung (1979))

c) Partitioning. Within one computer system there is a need for partitioning i.e. supplying interconnection between all the resources (usually i/o devices and memory) of each user, so that the subnetworks interconnecting the resources of different users are disjoint. (Thompson (1978), Masson, Gingher and Nakamura (1979), Goke and Lipovsky (1973)). A network which implements the partitioning of resources into disjoint subsets, interconnected by disjoint subnetworks is called a partitioner.

Switching networks were first investigated by people dealing with telephony. Hence most of the terminology derives from it (see for example Beneš(1965), Pippenger (1978a), Cantor(1971)).

Switching networks were built from lines and switches (or crosspoints). Each switch can be "closed" to enable transmission of information from an input line to an output line or it may be open to disable it (see figure 1.1). Many switching networks use crossbars as building blocks. A crossbar (a complete crossbar) has a switch between every input and every output line, a sparse crossbar has fewer switches (see figures 1.2a, 1.3a)

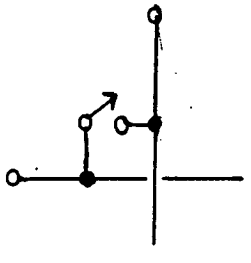


Figure 1.1
A switch

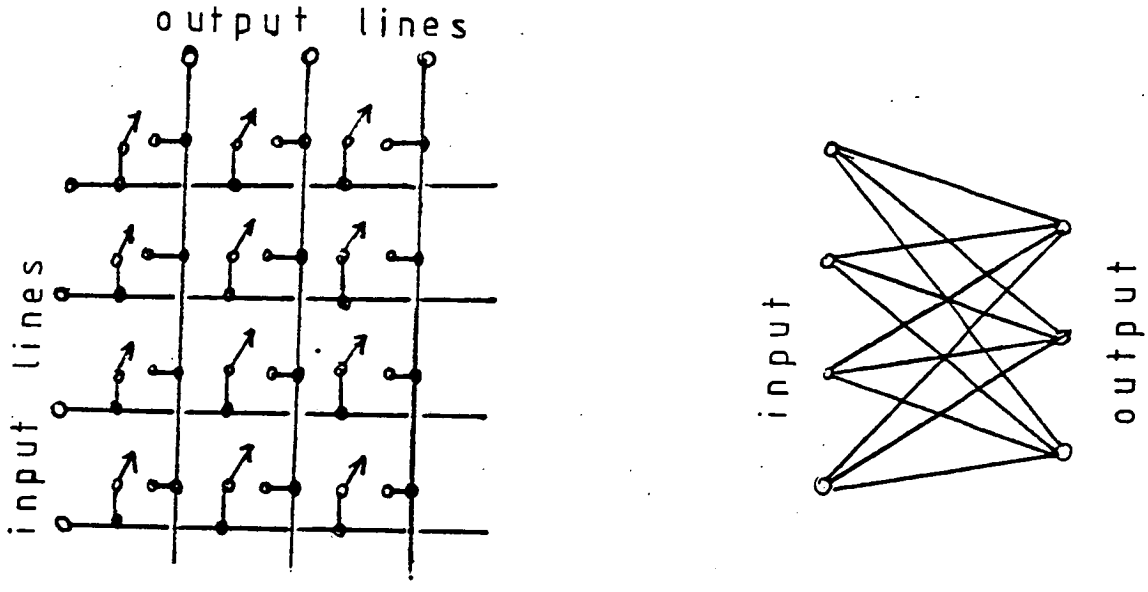


Figure 1.2 A 3x4 crossbar and the corresponding bipartite graph

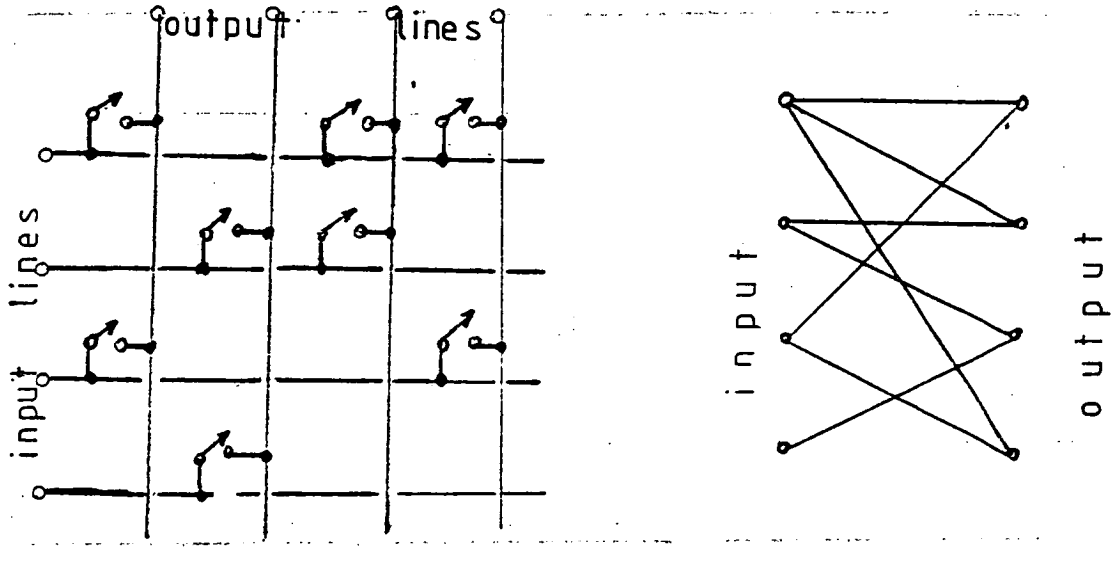


Figure 1.3 A sparse crossbar and the correspnding bipartite graph

Switching networks are usually represented by graphs in which each line corresponds to a node and each switch connecting two lines to an edge from an input node (of the switch) to an output node. (Hereafter we use the terms switch and edge equivalently) In this representation crossbars are represented by bipartite graphs (see figure 1.2, 1.3, 1.4))

In the switching networks discussed below, node disjoint paths from input nodes to output nodes are sought. They represent the closed switches connecting the input and output nodes. (A path is a sequence of nodes connected by a sequence of edges:

$$v_0 \overset{e_0}{\text{-------}} v_1 \overset{e_1}{\text{--------}} \ldots \qquad v_{n-2} \overset{e_{n-2}}{\text{-------}} v_{n-1} \quad)$$

For a given connection every path is a route from input to output; a node which is not on any route is an idle node.

The capacity of the network is the number of node disjoint paths it can implement. A network with N input nodes, M output nodes and capacity C is called an (N,M,C)-network. When N=M=C the network is an N-network.

1.3 Program execution graphs

The interest in networks representing program execution among computer scientists derives from the hope that these networks may help in understanding the inherent complexity of computing certain functions. These networks may be hard-wired networks for computing certain functions (see Batcher(1968), Knuth (1973, vol III)). They may represent the interdependence of the execution steps while computing a function. (Tompa (1978), Valiant (1975a),(1977))

For networks representing interdependence of execution steps only the networks of straight line programs will be considered. Straight line programs are programs which can be computed so that their network representation is acyclic. The networks representing program execution are therefore directed acyclic graphs with a set of specified input nodes and a set of specified output nodes. The input nodes are the nodes with indegree 0 and they correspond to the input variables (or constants). Each other node has indegree greater than 0 and it corresponds to a function

of its predecessors $Vk \leftarrow f(Vi_1,.....Vi_j)$. Therefore each non-input node is called a computed node (a terminology which will be used for switching networks as well). A computed node which is not an output node is an internal node. All the nodes with outdegree zero are output nodes, but there may be output nodes with outdegree greater then zero. The indegree of the nodes is bounded, and unless otherwise specified it is assumed to be 2.

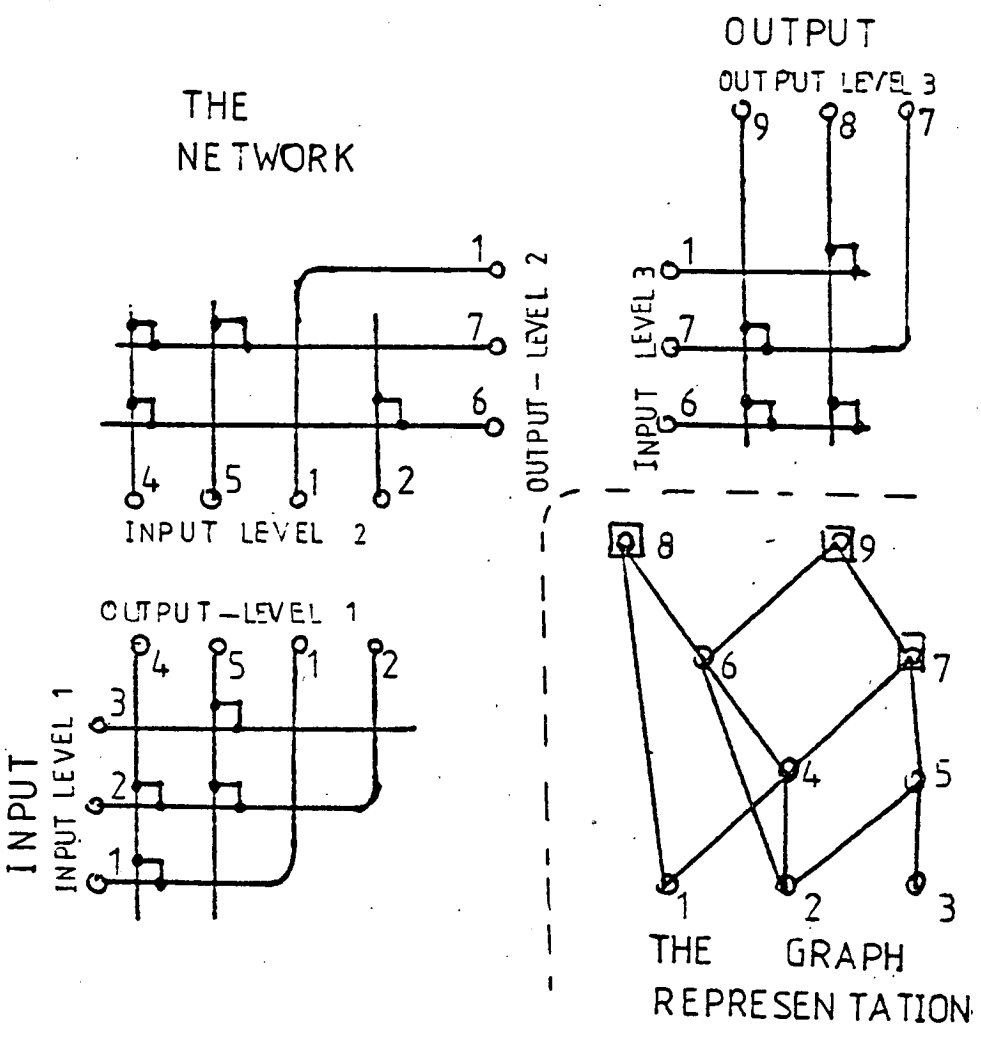
If for each node in the graph the function f is a linear combination of its arguments, i.e $f(u,v)=a\cdot u+b\cdot v$ $a,b \in F$ (a,b belong to the field over which the computation is performed) then the graph represents a linear program.

1.4 Size and depth of networks.

In switching networks the expensive components are often the switches. Therefore the aim is to construct networks with minimal number of switches, i.e, with minimal number of edges in the graph representation.

In computation networks each node represents a computation step and requires resources: logical gates in the case of boolean hardwired networks or cpu time for algorithms computed on one-cpu computer. The aim is to construct networks with minimal number of computed nodes.

In order to discuss parallel computation and time delays the nodes in a directed network are labelled, i.e assigned levels, as follows:
All the input nodes are at level 0.
Each computed node has a level greater than the levels of all of its predecessors.



A levelled connection network and its graph representation

Figure 1.4

In switching networks the switches at level i are those represented by edges incoming to nodes at level i (see figure 1.4). "Levels" are often called "stages" in the literature, but we shall use the latter term in the following extended sense: If a network is constructed from subnetworks so that the outputs of one set of subnetworks are the inputs to another set of subnetworks then each such set of subnetworks defines a stage in the system. When the subnetworks in each stage are crossbars (or sparse crossbars), the switches in stage i are also in level i. A consistent (not always optimal) level assignment is to assign to

each node the length of a maximal path (defined below) from an input node to it. In this case the level is the depth of the node from the input.

In combinational logic circuits all the gates corresponding to nodes at the same level can be set simultaneously. In a multiprocessor environment all the computations corresponding to nodes at one level can be performed simultaneously and in switching networks all the switches at one level can be set simultaneously, and all the switches in the same stage can be set in a constant time. The maximal level (or stage), i.e the depth of the network represents the parallel time required to compute the function or to transfer information from input to output. When considering parallel computations the aim is to reduce the depth of the network, which corresponds to parallel time, possibly at the expense of increasing the number of edges and nodes.

1.5 Concentrators, superconcentrators, hyperconcentrators, and permutation networks.

A network which finds usage in telephony, in transferring calls from subscribers to a trunk line, is a concentrator. formally: An (N,M,C)-concentrator is an N input M output network ($M \leq N$) which for every $k \leq C$ can connect by node disjoint paths any set of k given input nodes to some set of k output nodes. We are interested in the case C=M.

Pinsker(1973) who defined those networks, proved that concentrators of size 29N edges exist. Margulis (1973) gave a linear construction for concentrators that depended on a parameter which he could not determine. Gabber and Galil (1979) found a linear construction with a very large linear factor (over 200). Our constructions of hyperconcentrators imply constructions for concentrators of size $1.5N\log_2 N+N$ switches when N is a power of 2,

which is better than even the non-constructive upper bound of Pinsker for every $N \leq 2^{18}$. (For small size concentrators with special properties see also Masson(1977)).

A Superconcentrator (SC) is a network in which there are k mutually node disjoint paths between any set of k input nodes and any set of k output nodes. (The order of the connections is immaterial). In a Hyperconcentrator (HC) for each set of $k \leq C$ input nodes there are node disjoint paths to the first k output nodes.

Superconcentrators were defined by Valiant (1975a), who showed that there is a superconcentrator with 234N edges. Pippenger (1977a) improved his construction to 39N edges, and we slightly improve it to 38.5N edges. We also give a lower bound of 5N+o(N) edges and 4N+o(N) nodes for SCs. Gabber and Galil (1979) gave recently an explicit linear construction for SCs with 263N edges. We give a non linear construction for SCs with $3N\log_2 N$ edges, which improves on the explicit construction for every $N \leq 2^{87}$, and is better than the nonconstructive upper bound for $N \leq 2^{13}$.

Valiant (1975a) showed that given a linear program to compute the matrix $\underline{y}=\underline{x}A$, if a minor of A of size n is not singular, then there are n node disjoint paths from the n input nodes corresponding to the rows of the minor to the n output nodes corresponding to the columns of the minor. He showed as a corollary that any linear straight line program for convolving two degree $N-1$ polynomials is an N-SC, and any linear program for computing the Discrete Fourier Transform (DFT) is an N hyperconcentrator. Tompa (1978) showed that the linear program for computing the DFT is an ultraconcentrator (defined in chapter 2), and pointed out that as a consequence of Dieudonné's (1970) proof that all the minors of the NxN-DFT matrix over the complex field are non singular when N is a prime, the linear program to compute the DFT in this case is a SC. Our lower bound for SCs gives a lower bound of $4N-O(N \log N)^{1/2}$ for the number of additions in a linear program

computing the DFT of prime order. This is the best lower bound known for the number of additions in DFT excluding results for more restricted models. (see for example Morgenstern(1973)).

By examining small size SCs we managed to reduce the number of additions: for convolving three numbers from 6 to 5, for DFT of three numbers from 6 to 5, and for DFT of five numbers from 17 to 15. (The DFT algorithm compared was that of Winograd (1978), which gives fewer additions than the standard DFT algorithm.)

In contrast with our relative ignorance about the size complexity of the above described networks, when we come to permutation networks the asymptotic complexity is fairly well understood. A permutation network (an N-connection network) is a network which can implement by node disjoint paths all the N! assignments of input nodes to output nodes. The construction of O(NlogN) switches is usually attributed to Joel (1968) and Waksman (1968), though it was first suggested by Beizer(1962). The best construction of these networks using crossbar switches requires $6N\log_3 N+O(N\log N)^{1/2}$ switches (Pippenger 1978b)). That $6N\cdot\log_6 N-O(N)$ switches are required follows from a counting argument (Pippenger (1979)).

All the networks mentioned above and the others to be discussed later can be classified as being rearrangeable networks: when given a connection which is less than the capacity of the network and an idle input node (and an idle output node), it may be necessary to change (rearrange) the existing paths in order to add the node(s) to the connection.

1.6 Parallel algorithms

Before presenting our parallel routing algorithm, we shall give the model of parallel computation assumed by the algorithm. This is an extension of the RAC of Angluin and Valiant (1979). In this model the operation of all the processors is synchronized, and all the processors execute the same program.

Various models of parallel computation have been suggested in the past (Pratt and Stockmeyer (1976), Fortune and Wyllie (1978), Goldschlager (1978)). In those models the focus was on the ability of the machines to solve rather hard problems, for example NP complete problems. Therefore most of those models can use exponential numbers of processors in O(N) time. Our model restricts the number of processors to be polynomial in the input size. It is presented in order to make the parallel algorithms explicit. (We also point out how to translate an algorithm given in this model to a boolean network, increasing the depth of the network by a factor of O(logN) at most.)

Parallel algorithms presented in the literature can be divided into those which operate on a particular computer architecture (Kant and Kimura(1975), Thompson and Kung (1976)) and those which have an unrestricted parallel access to a global memory. (Hischberg (1976),(1978), Preparata(1978), Gavril(1975), Adjorandi and Corneil(1975)). Our algorithms fall into the second class.

The basic operation which makes our algorithms work in parallel, is a restricted kind of transitive closure operation which can be computed in O(logN) time rather than the familiar $O(\log^2 N)$ time. Using the information which is available in the particular problems it is possible to avoid the "fetch conflicts". A further feature of the algorithms is that they assume that each processor has a distinct priority, and this is used as an arbitrator, when multivalued functions are computed.

1.7 Edge colouring and routing

An efficient sequential routing algorithm for permutation networks built from three stages of crossbars, the first and the third stages being crossbars of capacity 2, was given by Waksman (1968). It can be generalised to three-stage networks when d is a power of 2 (Andersen(1977)). Applying this method recursively, it can give a routing algorithm in permutation networks built from crossbars with capacity a power of 2. No fast routing algorithm has been published for networks built from crossbars with capacity different from a power of 2. Translating the algorithms presented here into sequential algorithms, they can be made to run in $O(N \cdot d \cdot \log N)$ time or $O_d(N \cdot \log^2 N)$ time if d is not a power of 2.

Using N (or 2N) processors our routing algorithms find a routing in $O_d(\log_2^2 N)$ time if d is a power of 2, and in $O((d/\log_2 d)\log_2^3 N)$ or in $O_d(\log^3 N)$ time if d is not a power of 2. Note that for d=3, which gives a permutation network with minimal number of switches, both algorithms run in $O_d(\log^3 N)$ time.

The two routing algorithms are based on finding the routing using edge colouring. The first one (for d not a power of 2) is based on a parallel implementation of "typed recolour" of Gabow and Kariv (1978); the improved algorithm is based on "colour by pairs" of Gabow and Kariv (to appear). The edge colouring algorithm for d a power of 2 is based on creating paths in the graph (Gabow(1976)), which is done locally at each node, in parallel.

1.8 Thesis layout

In chapter 2 we present the known recursive construction of permutation networks. We give recursive constructions of superconcentrators, hyperconcentrators and various other networks of size $C \cdot N \cdot \log N$. For superconcentrators the value of C depends

on finding a superconcentrator of small enough size, whose existence is assured by the nonconstructive proof (Pippenger (1977a)). For hyperconcentrators it depends on finding a small ultraconcentrator, as hyperconcentrators are built recursively from ultraconcentrators. These constants are found in chapter 5.

We present a construction of generalized connection networks which are built from hyperconcentrators, hypergeneralizers and permutation networks. The improved value on the size of hyperconcentrators improves the size of this network as well.

At the end of the chapter we show that a network similar to the network for parallel prefix computation of Ladner and Fischer (1977) added to a permutation network can act as a partitioner of depth O(logN).

In chapter 3 a lower bound of 4N-o(N) and 5N-o(N) on the number of computed nodes and switches respectively is given for superconcentrators.

In chapter 4 the upper bound of 39N+o(N) edges (and 36N+o(N) computed nodes) derived by Pippenger is improved to 38.5N+o(N) edges (and 32.5N+o(N) computed nodes). The exact upper bound of 40N edges given by Pippenger is improved to 39.05N, using results from chapter 5.

In chapter 5 small size superconcentrators are examined. We give all the superconcentrators of size three with indegree two and minimal number of computed nodes (i.e. 5). We enumerate what we believe to be all the superconcentrators with indegree 2 and capacity 4 and 8 computed nodes (though we did not succeed in proving that this list is exhaustive). We show that by examining the size-3 superconcentrators, which are also hyperconcentrators, we can reduce the number of additions in convolutions and DFT of 3 numbers, and in the DFT of 5 numbers.

Chapter 6 brings the model of parallel computation used by the algorithms in the following chapters.

In chapter 7 the parity-labelling algorithm, which is the basis for the edge colouring algorithms and the routing algorithm, is presented. We discuss the importance of what we call "the orientation", (defined there), and show that if missing, finding it may require $N^2/\log_2 N$ prooessors and $O(\log_2 N)$ time.

Chapter 8 presents the edge colouring algorithms.

Chapter 9 gives the routing algorithm, and shows how it may be implemented for recursively constructed superconcentrators, as well as for permutation networks.

In chapter 10 we give parallel algorithms for finding a maximal (not maximum!) matching in bipartite graphs, in $O(\log^4 N)$time.

2 RECURSIVE CONSTRUCTION OF NETWORKS

2.1 Introduction

Recursive constructions of various networks are presented in this chapter. First, the standard recursive construction of Permutation Networks (PN) is reviewed (Beneš (1965), theorem 3.1, Waksman (1968)). Then, recursive constructions of Superconcentrators, (SC), Hyperconcentrators, (HC) Supergeneralizers (SG) and some other networks are developed. In each case the size of the resulting networks is CNlogN. Linear non-constructive upper bounds for these networks are already known, but these have rather large multiplicative constants; An explicit linear construction with even larger constants was recently given by Gabber and Galil (1979); - Our aim here is to give constructions which are smaller than the previous explicit constructions for all reasonable size networks.

For SCs, the best known non constructive linear upper bound is 38.5N edges (see chapter 4). The explicit linear construcion of Gabber and Galil(1979) yields SCs of size 263N edges. Our construction achieves an upper bound of $3N\log_2 N$ edges, as shown in chapter 5, which is better than the linear explicit construction for all $N \leq 2^{87}$. It also improves on the existential bound for $N \leq 2^{13}$. Our construction is better by about 20% than the recursive construcion of PNs, which till now has been the best construction known for SCs. Furthermore, the constant multiplier can be improved in principle, by finding suitably small N-SCs for appropriate fixed N.

For HCs, the construction achieves an upper bound of $(3/2)N\log_2 N$. By deleting output nodes, a HC can serve as a concentrator. This construction improves on the known constructions of concentrators for all reasonable size networks.

A Generalized Permutation Network, GPN, (called elsewhere (Pippenger 1977b, Thompson 1978)a Generalized Connection Network) is a network which can implement by node disjoint trees any one to many assignment from input nodes to distinct output nodes. Pippenger (1977b) has proved that such a network requires at most a constant times N more edges than a PN. The best practical construction known, however, uses CNlogN more edges. Such a construction is based on concatenating a Supergeneralizer (SG, defined later) to a PN, where a SG can be constructed by concatenating a HC and a Hypergeneralizer (HG). Such a construction, using Order-preserving networks is described in Thompson (1978), who improved on a network of Ofman (1967). It is shown in this chapter that it is possible to eliminate the order preserving property of the HCs, but we did not succeed in eliminating the Order preserving property in the construction of the HG. This gives a small improvement in the multiplicative constant over the best construction known for all reasonable values of N.

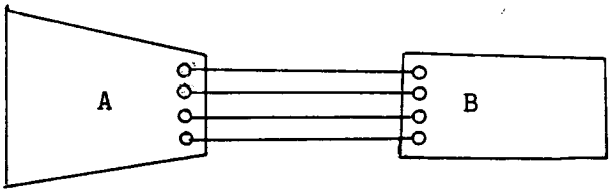
To summarize, SCs can be constructed recursively in terms of smaller SCs (theorem 2.2). Ultraconcentrators can also be constructed in terms of themselves (theorem 2.3). For Hyperconcentrators we use both UCs and HCs as components (theorem 2.4). SGs are obtained by concatenating a HC to a HG (theorem 2.8). The latter are obtained by constructing order preserving ultrageneralizers (OUGs) recursively (theorem 2.5), and using OHGs at the last stage (theorem 2.6).

In section 2.7 it is shown that in order to construct a partitioner it is enough to concatenate a PN and a Generalizer (as defined in this chapter), without needing the more powerful HG (Thompson 1978). It is shown that a network similar to Ladner and Fischer's (1977) prefix computation network is a Generalizer with linear size and $2\log_2 N$ depth.

2.2 Definitions

Let $A(N_A,M_A)$ be a network with N_A input nodes and M_A output nodes, and similarly $B(N_B,M_B)$.

If $M_A = N_B$ the concatenation of A and B = A·B is the network in which each output node of A is identified with a distinct input node of B (see figure 2.1).



The lines denote identification of nodes

Figure 2.1 A·B

The Product of A and B AxB is constructed by taking N_B copies of A and M_A copies of B and identifying each output node of the A s with a distinct input node of the B s, so that each of the output nodes in one copy of A is identified with an input node in a different copy of B (see figure 2.2).

Let (nA,j) be the j'th node in the n'th copy of A, and let input(nA,j), output(nA,j) be the j'th input /output node in the n'th copy of A. For ease of discussion it is assumed that in AxB output(nA,j) is identified with input(jB,n). Different connections can be implemented by different indexing of the output nodes of A.

a(i), b(i), p(n,j). These functions are defined relative to a given partition of N numbers. The partition's parameters d or d,q are usually omitted.

For N=d·q (d divides N) and for N=d·q+r, r<q

$a_d(i) = \lfloor i/d \rfloor \quad b_d(i) = i \bmod d \quad p_d(n,j) = d \cdot n + j$

For N=dq+(d-1)s (N is partitioned to q blocks of size d and s blocks of size d-1 each)

$$a_{d,q}(i) = \begin{cases} \lfloor i/d \rfloor & \text{if } i < dq \\ q + \lfloor (i-dq)/(d-1) \rfloor & \text{if } i \geq dq \end{cases}$$

$$b_{d,q}(i) = \begin{cases} i \bmod d & \text{if } i < dq \\ (i-dq) \bmod (d-1) & \text{if } i \geq dq \end{cases}$$

$$p_{d,q}(n,j) = \begin{cases} n \cdot d + j & \text{if } n < q \\ dq + (n-q)(d-1) + j & \text{if } n \geq q \end{cases}$$

For example the i'th input node of the graph AxB is input(a(i)A,b(i)) of AxB, and input(nA,j) is the p(n,j)'th input node of the graph.

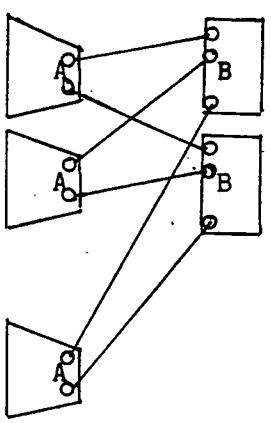


Figure 2.2 AxB

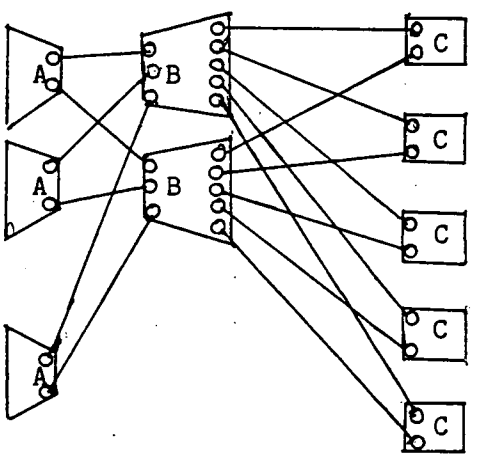


Figure 2.3 (A,B,C)

Given A,B,C, where $M_A = N_C$, the triple product (A,B,C) is constructed by taking N_B copies of A, M_B copies of C and $M_A = N_C$ copies of B, and identifying the output nodes of the A s with the input nodes of the B s as in AxB and identifying the outputs of the B s with the inputs of the C s as in BxC (see figure 2.3).

A^* A reversed is derived from A by reversing the direction of

each arc in A. The input nodes of A^* correspond to the output nodes of A, and the output nodes of A^* correspond to the input nodes of A.

$\underline{AxxB = (A,B,A^*)}$

A bipartite graph is a graph in which the nodes can be partitioned into two groups A,B so that all the edges are between nodes in A and nodes in B. There are no edges between nodes of the same group. It is possible to refer to one of the groups, say A, as the input nodes and to the other group as output nodes, thus assigning to the edges a direction from A to B.

An edge colouring of a graph is an assignment of colours to the edges, so that two edges incident with the same node are coloured in different colours.

In this chapter and in chapter 5 the following result is used repeatedly:

Lemma 2.1

For N a power of d

the recurrence $F(N)=x\cdot\frac{N}{d} - y + d\cdot F(\frac{N}{d})$ satisfies

$$F(N) = \frac{x}{d} N\log_d N - \left(\frac{y}{d(d-1)} + \frac{x-F(d)}{d}\right)N + \frac{y}{d-1}$$

Proof

Can be verified by substitution.

2.3 Permutation networks

An N-Network is a network with N input nodes and N output nodes.

An N-Permutation Network, N-PN, is an N-network which can implement by node disjoint paths all the N! assignments of input nodes to output nodes.

Let $\Sigma = \{(i,c(i))\}$ be a connection assignment. We say that input node i is associated with output node c(i).

Given a permutation network and a partial permutation assignment, an input node is an active input node if it is associated with some output node by the permutation. Any such output node is an active output node. If an implementation of the permutation is given, then any node on a route of the implementation is an active node also.

Theorem 2.1 (Beneš 1965 and Waksman (1968))

If A and C are d permutation networks and B is an (N/d) PN then (A,B,C)-A is an N PN. [Here (A,B,C)-A is derived from (A,B,C) by identifying the input and output nodes in one copy of A and erasing all its edges, i.e. by erasing one copy of A.]

Proof

The proof shows that each connection request which is a subassignment of a permutation, can be implemented by the network, and establishes a routing corresponding to the request. It is different from the proof of Beneš or Waksman, but gives a routing strategy that is easily implemented by the parallel algorithm of chapter 9.

1) Let $\Sigma = \{(i,c(i))\}$ be the connection requirement.

2) Construct a bipartite graph G with N input and output nodes, corresponding to the input and output nodes of (A,B,C) respectively. For each pair of associated nodes construct an edge from input node i to output node c(i) and denote it by (i,c(i)).

3) From G construct a bipartite graph G' by identifying all the input nodes that belong to the same copy of A and all the output nodes that belong to the same copy of C. (All the nodes with the same value of a(i) are identified).

   G' is a bipartite graph with maximal degree d (see figure 2.4a,b).

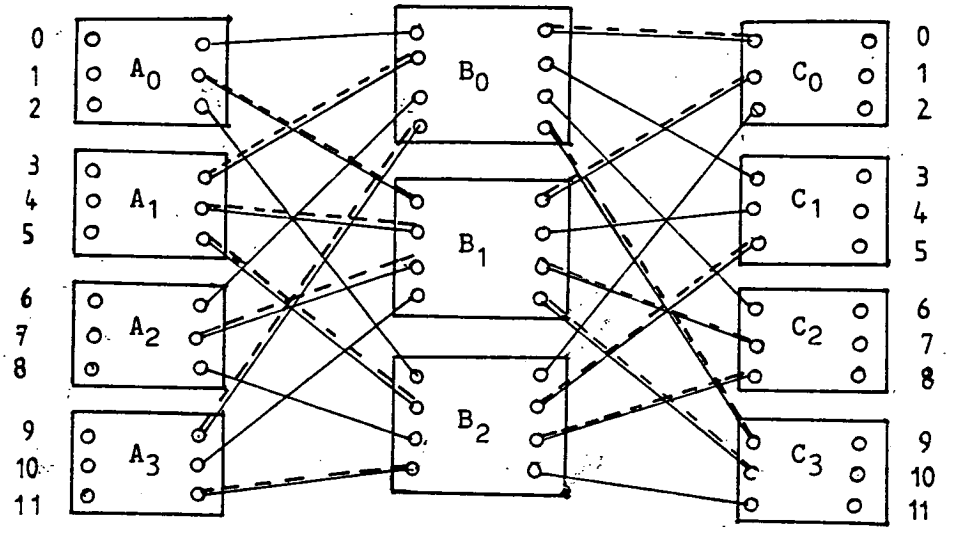
4) It is a well known result that G' can be coloured in d colours. (Berge (1973), theorem 2 p 250, derived from corollary 4 p 135). Moreover, G' can be coloured in d colours when the colours of the edges incident with one node are predetermined. Find a d colouring for G' in which the edges (i,c(i)) incident with the eliminated copy of A (if such edges exist) are coloured b(i).

5) For each edge in the bipartite graph, if the edge (i,c(i)) is coloured k, it is possible to establish a connection from input node i to output node c(i) through the k'th copy of B.

   Colour k is assigned to nodes i and j if the edge (i,j) is coloured k.
   All the active input nodes in the same copy of A are assigned different colours, so the colouring establishes a connection which can be embedded in a permutation in each copy of A. It can be implemented by node disjoint paths, as A is a PN. Similarly for each copy of C.

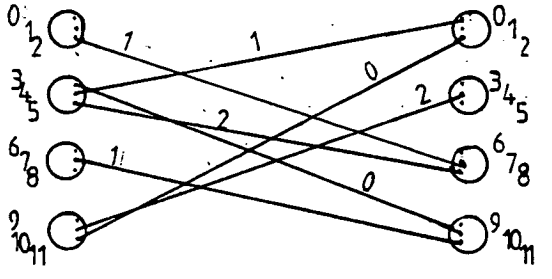
   Input node i is input(a(i)A,b(i)). In the a(i)'th copy of A

there is a path from input(a(i)A,b(i)) to output(a(i)A,k), which is identified by the construction with input(kB,a(i)). B is a permutation network, therefore it is possible to establish in the k'th copy of B a path from input(kB,a(i)) to output(kB,a(c(i))), which is identified with input(a(c(i))C,k). Finally it is possible to establish a path from input(a(c(i))C,k) to output(a(c(i))C,b(c(i))), thus completing the path from i to c(i).



A,C are d size networks and B is an (N/d)-network.
The full lines denote identification of nodes, the dashed ones are solution to the assignment
{(2,7),(3,10),(4,0),(5,8),(7,11),(10,4),(11,2)}

Figure 2.4a Routing in (A,B,C), a PN/SC



The corresponding bipartite graph and its colouring

Figure 2.4b

Let 'netname'_S(N) denote the minimal (possible, or in a particular construction)(*) number of switches (edges) in a network of size N, and let 'netname'_N(N) denote the (minimal) number of computed nodes in a network of size N where the indegree of the nodes is limited to 2.

Corollary 2.1

$PN\_S(N) = 6N\log_3 N-(9/2)N+9/2$ for N a power of 3

$PN\_S(N) \leq 6N\log_3 N+o(N\log N)$ for all N.

$PN\_N(N) \leq 2N\log_2 N+o(N\log N)$.

Proof

Using the result of theorem 2.1, it is possible to build PNs recursively: PN(d)xxPN($\lceil N/d \rceil$) is a PN(N)

$$PN\_S(N) \leq \left\lceil \frac{N}{d} \right\rceil \cdot PN\_S(d) - PN\_S(d) + d \cdot PN\_S\left(\left\lceil \frac{N}{d} \right\rceil\right)$$

Which gives for N a power of d, using lemma 2.1,

$$PN\_S(N) \leq \frac{2 \cdot PN\_S(d)}{d}N\log_d N - \frac{PN\_S(d)}{d}\left(\frac{1}{d-1} + 1\right)N + \frac{PN\_S(d)}{d-1}$$

and for each N

$$PN\_S(N) \leq 2(PN\_S(d)/d)N\log_d N+o(N\log N).$$

Substituting d=3, 3-PN a 3x3 crossbar; gives the best constant multiplier:

$PN\_S(N)=6N\log_3 N-(9/2)N+9/2$ if N is a power of 3,

$PN\_S(N) \leq 6N\log_3 N+o(N\log N)$ for all N.

[When d=2 and PN(2) is a 2x2 crossbar, the known permutation network of Waksman(1968) gives a higher multiplier:

$PN\_S(N) \leq 4N\log_2 N+o(N\log N)=6.339N\log_3 N+o(N\log N)$.]

When the number of computed nodes with indegree 2 is considered, a PN built from 2x2 crossbars satisfies $PN\_N(N) \leq 2N\log_2 N+o(N\log N)$.

Pippenger (1978b) has shown that the number of switches in a PN

(*) The context will make clear which meaning is intended.

built recursively from crossbars, (without deleting a copy of A at each stage), assuming different size crossbars at each stage of the construction, is minimized when it is built from 3x3 crossbars, the construction repeated $k=\lfloor \log_3 N-(1/2)\log_3\log_2 N+\log_3(3/2) \rfloor$ times and the middle stage of the network is formed from MxM crossbars for $M=\lceil N/3^k \rceil$. This network has $6N\log_3 N+O(N(\log N)^{1/2})$ switches.

2.4 Superconcentrators (SC)

Theorem 2.2

Let A,C be d-SCs and let B be an $\lceil N/d \rceil$-SC.

a) If d divides N then (A,B,C) - A is an N-SC.

b) Otherwise, let N=qd+s, let B' be an $\lfloor N/d \rfloor$-SC, let C' be an s-SC and let A' contain s nodes, each one an input node identified with an output node.
(A,B,C)-(d-s)·B+(d-s)·B'+C'+A' is an N-SC, where each input node in C' is identified with an output node of the B s and each node of A' is identified with an input node of the B s.

This SC can be represented schematically as

C...($\lfloor N/d \rfloor$ times) C C'
B... (s times) B B'...(d-s times) B'
A... ($\lfloor N/d \rfloor$ times) A A'

Proof

The proof follows the proof of theorem 2.1.

An order preserving assignment is an assignment in which for each i<j the output node(s) associated with input i precedes the output node(s) associated with input j.

1) For each superconcentration requirement establish an order preserving assignment of the active input nodes to the active

output nodes. Let $\Sigma = \{(i,c(i))\}$ be the connection assignment. Clearly, if there are s_1 active input nodes in A' and s_2 active output nodes in C', $\min(s_1,s_2)$ nodes in A' are associated with nodes in C'. [N.B. For case (a) the order preserving restriction is not necessary.]

2) Construct a bipartite graph G with an edge corresponding to each pair of associated nodes in the assignment.

3) Construct G' from G by identifying all the nodes with the same value of a(i).

4) Find an edge colouring of G' in which the edges (i,c(i)) incident with the missing copy of A (or with A') are coloured b(i). If $s_2 > s_1$, find such a colouring in which, in addition, the edges incident with C' are from the first s colours.

5) For each n, let the active input nodes in the n'th copy of A $i_1,\ldots i_r$, be "coloured" $k_1,\ldots k_r$ respectively. Since A is a SC there are in A r node disjoint paths from $b(i_1),\ldots b(i_r)$ to nodes numbered $k_1,\ldots k_r$, not necessarily from input node i_j to output node k_j. Similarly, in each copy of C there are node disjoint paths to the active output nodes from the input nodes corresponding to the colours assigned to the edges incident with the supernode.

For each active input node i, such that (i,c(i)) is coloured k, there is a path from some input node in the a(i)'th copy of A to output(a(i)A,k), which is identified with input(kB,a(i)). Output(kB,a(c(i))) is identified with input(a(c(i))C,k), from which there is a path to some output node in the a(c(i))'th copy of C. Therefore for each edge coloured k there is an active input node and an active output node in the k'th copy of B.

By the recursive construction B is a SC, and the number of

active input nodes in each copy of B equals the number of active output nodes, therefore it is possible to establish node disjoint paths from the active input nodes in B to the active output nodes in B.

Previous recursive constructions of SCs used PNs as building blocks. Theorem 2.2 proves that it is possible to build SCs recursively from smaller SCs.

Corollary 2.2

For $N=d \cdot q$

1) $SC^{-}S(N) \leq \frac{2N}{d} \cdot SC^{-}S(d) - SC^{-}S(d) + d \cdot SC^{-}S(\frac{N}{d})$

which gives for N a power of d

.2) $SC^{-}S(N) \leq \frac{2 \cdot SC^{-}S(d)}{d} N \log_d N - \frac{SC^{-}S(d)}{d}(\frac{1}{d-1} + 1)N + \frac{SC^{-}S(d)}{d-1}$

For $N=dq+r$, $r<d$

.3) $SC^{-}S(N) \leq \frac{2N}{d} SC^{-}S(d) + SC^{-}S(r) + r \cdot SC^{-}S(\lceil \frac{N}{d} \rceil) + (d-r) \cdot SC^{-}S(\lfloor \frac{N}{d} \rfloor)$

which gives for all N

.4) $SC^{-}S(N) \leq \frac{2 \cdot SC^{-}S(d)}{d} \cdot N \log_d N + o(N \log N)$

Proof

Obvious, using the result of theorem 2.2.

In order to make comparisons with permutation networks, it is convenient to express the above bounds as follows:

$$SC^{-}S(N) \leq \frac{2 \cdot SC^{-}S(d)}{d} \cdot N \log_d N + o(N \log N) = 6N \log_3 N \frac{SC^{-}S(d)}{3d \cdot \log_3 d} + o(N \log N)$$

Similarly, the minimal number of computed nodes with indegree 2

satisfies:

$$SC^{-}N(N) \leq \frac{2 \cdot SC^{-}N(d)}{d} \cdot N\log_d N + o(N\log N) = 2N\log_2 N \frac{SC^{-}N(d)}{d \cdot \log_2 d} + o(N\log N)$$

In chapter 5, SCs which satisfy $\frac{SC^{-}S(d)}{3d\log_3 d} < 1$ are given.

This results in SCs of size smaller than the size of the corresponding PNs. It is shown there that (theorem 5.1, 5.2) for all N

.5) $SC^{-}S(N) \leq 3N\log_2 N - (5/4)N + 2$

and for N a power of 2

.6) $SC^{-}S(N) \leq 3N\log_2 N - 3N + 3$

No explicit SC which satisfies $\frac{SC^{-}N(d)}{d \log_2 d} \leq 1$ is known to me, though the existential proof guarantees that they exist.

2.5 Hyperconcentrators (HC) and Ultraconcentrators (UC)

The first theorem in this section gives a recursive construction of UCs. HCs can then be constructed from UCs and fixed size HCs, as described in the second theorem.

An (N,M,C)-Ultraconcentrator (UC) is an N-input, M-output network, in which for each set of $K \leq C$ input nodes there are K node disjoint paths to any set of K output nodes, that are consecutive in circular order.

In a circular order node 0 follows node N-1. If the active nodes are consecutive, and if there are idle nodes, then node i is the first active node if node i-1 is idle (and i is active). If there are no idle nodes each node can be chosen to be the first active node.
For nodes in circular order

$$i+j = \begin{cases} i+j & \text{if } i+j<N \\ i+j-N & \text{if } i+j\geq N \end{cases}$$

Theorem 2.3

If A is a d-UC and B is an (N/d)-UC, then AxB is (or, in the terminology of Cantor (1971), is isomorphic to) an N-UC.

Proof

Let C be a network consisting of d separate nodes, each node being an input and output node.
AxB is isomorphic to (A,B,C). C does not contribute to the resulting network because all its nodes are identified with the output nodes of the B s. We prove that (A,B,C) is an UC.

1) Let i_1 be the first active input node (in node index order), and let m be the first active output node (in circular order). Associate the j'th active input node with the j'th active output node.

2,3) Follow the proof of theorem 2.1 (see figure 2.5).

4) Assign to edge (i,c(i)) colour b(c(i)). All the edges incident with one copy of C are assigned different colours. All the edges incident with a supernode corresponding to one copy of A are assigned circular consecutive colours, therefore they are assigned different colours.

5) Colour k corresponds to a path through the k'th copy of B.

We show that the active output nodes in each copy of B are consecutive in circular order.
Suppose that in B_k, say, the active output nodes are not consecutive in circular order. Then, without loss of generality $\exists j_1<j_2<j_3<j_4$ with j_1, j_3 active and j_2, j_4 passive.

But those nodes are identified with $j_1|C|+k < j_2|C|+k < j_3|C|+k < j_4|C|+k$ respectively, implying that the active output nodes are not consecutive in circular order, contradicting the definition of an ultraconcentration requirement.

For each ultraconcentration requirement, the colours assigned to each copy of A are consecutive. A is an UC, therefore it is possible to find node disjoint paths from its active input nodes to the consecutive output nodes corresponding to its colours. The number of active input and output nodes in each copy of B is equal, and the active output nodes in each copy of B are consecutive, therefore it is possible to establish node disjoint paths from the active input nodes in each copy of B to the active output nodes, thus completing the connection of input nodes to output nodes in (A,B,C).

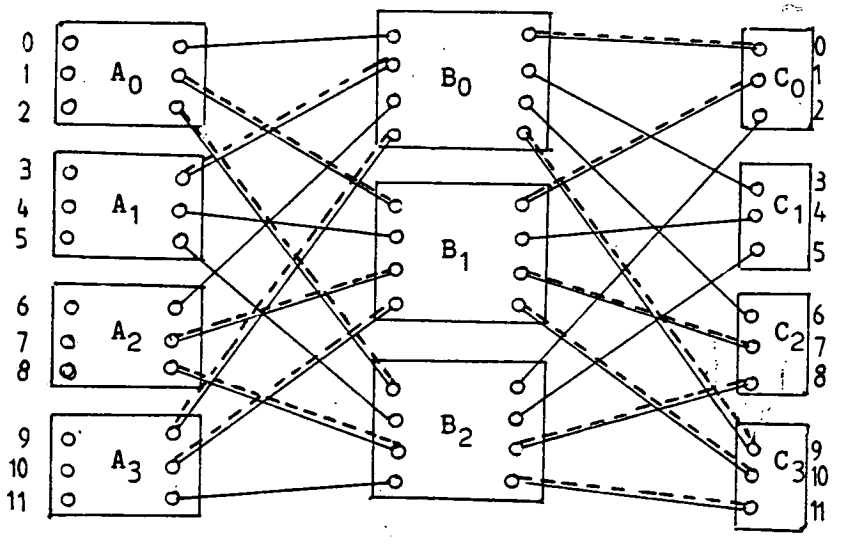
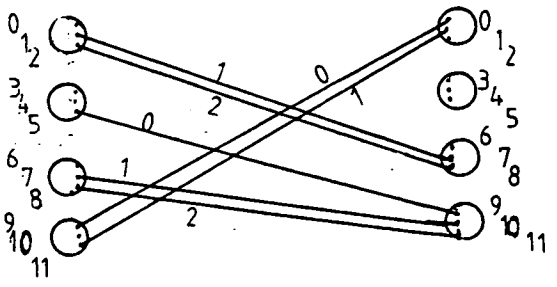


Figure 2.5a Construction of an UC (A,B,C) and routing in it.

The full lines denote identification of nodes.
If the active input nodes are 1,2,5,7,8,9,11 and the active output nodes are 7,8,9,10,11,0,1 (7 is the first one) the dashed lines indicate the routing.



The corresponding bipartite graph and its colours

Figure 2.5b

Corollary 2.3

$UC^{-}S(N) \leq (UC^{-}S(d)/d)N\log_d(N)+o(N\log N)=$

.7) $3N\log_3 N(\frac{UC^{-}S(d)}{3d\log_3 d})+o(N\log N)$

Proof

From theorem 2.3

$UC^{-}S(N) \leq \lceil N/d \rceil \cdot UC^{-}S(d)+d \cdot UC^{-}S(\lceil N/d \rceil)$

which has a solution

$UC^{-}S(N) \leq (UC^{-}S(d)/d)N\log_d N+o(N\log N)=$

.7) $3N\log_3 N(\frac{UC^{-}S(d)}{3d\log_3 d})+o(N\log N)$

The best recursive construction known so far was by using 3x3 crossbars, giving UCs of size

$3N\log_3 N+o(N\log N)$.

In chapter 5 it is shown that

.8) $UC^{-}S(N) \leq (3/2)N\log_2 N+o(N\log N)$.

Similarly the construction yields for the number of computed nodes

$UC^{-}N(N) \leq (UC^{-}N(d)/d)N\log_d N+o(N\log N)=$

$N\log_2 N \cdot UC^{-}N(d)/(d\log_2 d)+o(N\log N)$.

No UC which satisfies $UC^{-}N(d)/(d \cdot \log_2 d) < 1$ is known to me.

Theorem 2.4

If A is a d-UC, A' is a d-HC and B is an (N/d)-HC, then $AxB-A_0+A'_0$ is an N-HC. (The first copy of A, indexed zero, is replaced by A'.)

Proof

The proof follows the proof of theorem 2.3. It is only needed to show that for each hyperconcentration requirement, the resulting requirement in B is for hyperconcentration, and not for ultraconcentration.

If output(kB,i) is active, then input=output(iC,k) is active, Each $j<p(i,k)$ is an active output node and in particular for each $i'<i$, (i'C,k) is an active output node. This node is identified with output(kB,i'), therefore for each $i'<i$ output(kB,i') is an active output node, and the connection requirement in each copy of B is for hyperconcentration.

In the first copy of A the colouring assigns to the active input nodes the first colours, because the requirement is for hyperconcentration, and results in a hyperconcentration requirement.

Corollary 2.4

$HC^{-}S(N) \leq (HC^{-}S(d)/d)N\log_d N+o(N\log N)$.

Proof

From Corollay 2.3, as each UC is a HC.

In theorem 5.3 it is shown that $HC^{-}S(N) \leq 1.5N\log_2 N+o(N\log N)$.

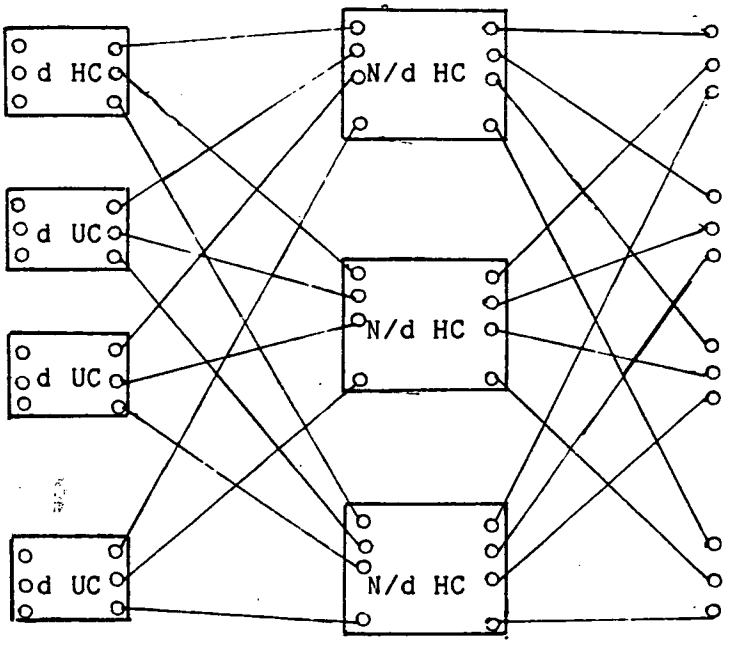


Figure 2.6 recursive construction of a HC.

This section showed that the multiplicative factor in the recursive construction of HCs is the one achieved in the construction of UCs. Only a small linear improvement, is achieved for HCs. This construction of HCs can serve to construct concentrators, by deleting any superfluous output nodes. It yields the best known construction for small size concentrators.

2.6 Generalizers, Supergeneralizers, Ultrageneralizers..

An N-Supergeneralizer, SG, (called elsewhere Generalizer, (Pippenger 1977b, Thompson 1978) but not to be confused with what is defined here as a generalizer), is a network which for each given set $i_1, i_2 \ldots i_r$ of input nodes and numbers $k_1\ k_2 \ldots k_r$, $\sum k_j \leq N$, can implement a connection from input node i_j to k_j different output nodes, by node disjoint trees.

An N-Ultrageneralizer UG is a network which for each set of r consecutive (in circular order) input nodes $i_1, i_1+1, \ldots i_1+r-1$ and a set of numbers $k_1, \ldots k_r$, $\sum k_j \leq N$, can implement by node disjoint

trees, a connection from input node i_j to k_j different output nodes.

An N-Hypergeneralizer HG is a restriction of an N-UG in which the first active input node is always node zero.

A Generalizer G, is a network which for a given set of numbers $k_1,\ldots k_r$, $\sum k_j \leq M$, can implement by node disjoint trees a connection from some input node to k_1 output nodes, from some other input node to k_2 output nodes...

An Order-preserving network is a network which can implement an order preserving connection. Each Order-preserving network will be prefixed with an O. In OSG,OUG the output nodes are in index order, the input nodes can be in circular order. In OSC,OUC the input nodes are in index order, the output nodes in circular order. Let O_G stand for any of OHG, OUG, OSG, and similarly O_C.

It is possible to implement in an O_G any assignment in which the output nodes to be connected to each input node are completely determined, by considering the assignment in which there are no idle output nodes, and then ignoring connections to unwanted output nodes.

Theorem 2.5

If A is a d-OUG and B is an N/d-OUG then BxA is isomorphic to an N-OUG.
Let C be a d-network of isolated nodes, each one an input and output node. We prove that (C,B,A) is an OUG.

Proof

The proof shows that each order-preserving ultrageneralization assignment in (C,B,A) results in such an assignment in each copy of B and A. A and B can implement the sub-assignments as they are

OUGs.

Input node i is (a(i)C,b(i)) and it is identified with input(b(i)B,a(i)). For each output node o that should be connected to input node i connect input(b(i)B,a(i)) to output(b(i)B,a(o)) which is identified with input(a(o)A,b(i)). Connect input(a(o)A,b(i)) to output(a(o)A,b(o)) to complete the path from i to o.

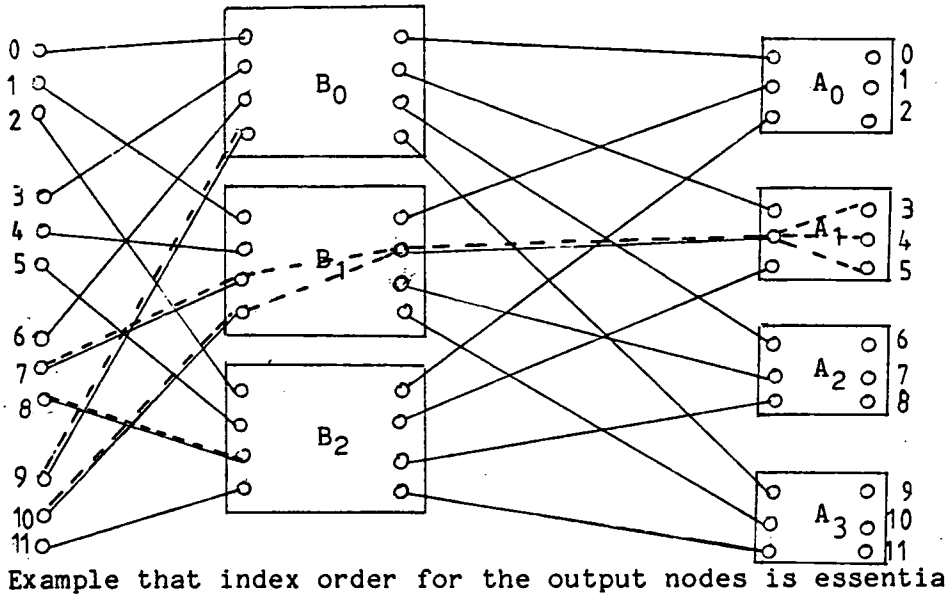
It is needed to show that the above assignment results in an OUG assignment in each of the subnetworks.

a) The active input nodes in each copy of B are consecutive and preserve the order.
If i,j are connected to the output through the same copy of B, say k, then b(i)=b(j)=k, because i = (a(i)C,b(i)) = input(b(i)B,a(i)). i<j (in circular order), therefore a(i)<a(j) (circular order). Each i', i<i'<j, is an active input node by the assignment, therefore for each a(i)<t<a(j) (tC,k) which is identified with input(kB,t) is active.

b) The assigned requirement in each copy of B is for Orderpreserving Ultrageneralization, and each pair of active input nodes is associated with distinct output nodes.
Assume i, j are to be connected to the output via the same copy of B, and let O'_i be the last output node to be connected to i and O_j the first one connected to j.
b(i)=b(j), i<j, therefore a(i)<a(j), $i \leq j-d$.
Each active input node is connected to at least one output node, therefore $O'_i \leq O_j-d$ and $a(O'_i) < a(O_j)$.

c) The input nodes in each copy of A are consecutive and preserving the order.
If $(a(O_i)A,b(i))$ and $(a(O_j)A,b(j))$ are active input nodes in the same copy of A, then $a(O_i)=a(O_j)$ and b(i) # b(j). If $O_i<O_j$, then by the definition i<j (circular order) and each

i<t<j is an active input node. j-i<d, therefore for each i<t<j t-i<d and it can be connected through the b(t)'th copy of B.



Example that index order for the output nodes is essential
Active input nodes 7,8,9,10 output nodes 5-7,8,9-2,3-4

Figure 2.7 routing in an OUG

Corollary 2.5

$$OUG^{-}S(N) \leq 3N\log_3 N+o(N\log N)$$

Proof

Using a 3x3 crossbar as A in the recursive construction implied from theorem 2.5 gives:
$OUG^{-}S(N) \leq 3N+3\cdot OUG^{-}S(\lceil N/3 \rceil) \leq 3N\log_3 N+o(N\log N)$.

Theorem 2.6

If A is a d-OUG, A' is a d-OHG, B is an (N/d)-OHG and C is a d-network of isolated-nodes, then
$(C,B,A)\text{-}A_0+A'_0$, isomorphic to $BxA\text{-}A_0+A'_0$, is an N-OHG.

Proof

It is needed to show only that each hypergeneralization requirement results in such a requirement in each copy of B and in the first copy of A.

If input(kB,j) is active, this is (jC,k), and each $q<p(j,k)$ (index order) is active, in particular for each $j'<j$ $p(j',k)$ is active and input(kB,j') is active.

The first active input node is node 0, it is identified with input(0B,0), and connected to output(0B,0). Therefore the requirement in the first copy of A is for hypergeneralization.

Theorem 2.7

a. (Thompson 1978) If A is an O_G then A* is the corresponding O_C.
b. The upper and lower bound for the number of switches ih an N-OUC and in an N-OUG is $3N\log_3 N+o(N\log N)$
c. The upper and lower bound for the number of switches in an OHC and in an OHG are $O(N\log N)$, and $\Omega(N\log N)$ respectively.

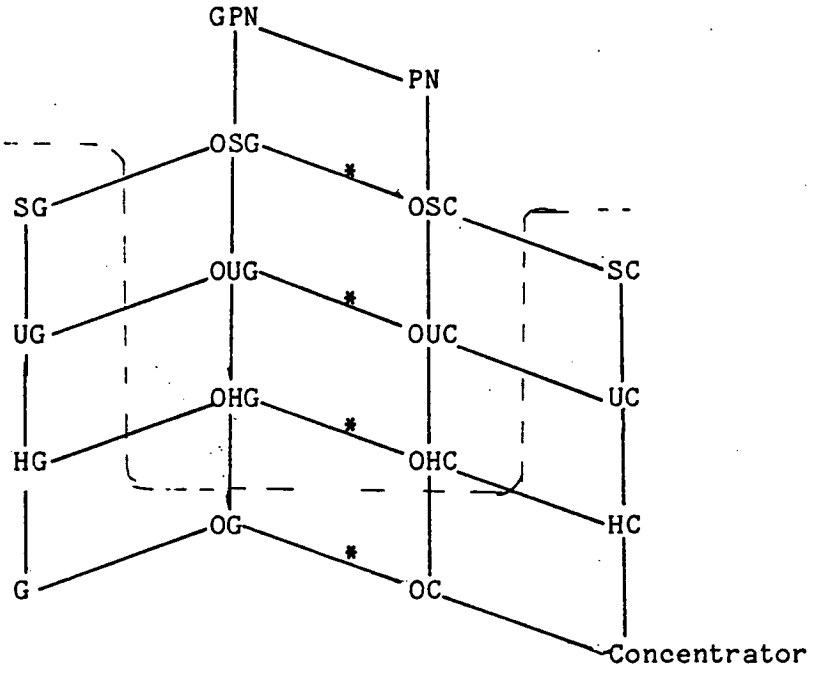
Proof

a. For each concentration requirement in O_C of input nodes $i_1<i_2<\ldots i_r$ (index order) to output nodes $j_1<j_2<\ldots j_r$ (circular order in OUC,OSC), consider the generalization assignment from input node j_k to output nodes $i_{k-1}+1$ through i_k, and then ignore all the output nodes different from $i_1,i_2,\ldots i_r$.

b. Each OUC can implement all the circular shifts. It is a known result that the lower and upper bound on the number of edges in shifting networks is $3N\log_3 N+o(N\log N)$ (Valiant 1975a), therefore the lower and upper bounds on the number of edges in OUCs and OUGs is $3N\log_3 N+o(N\log N)$, and the best shifting

networks, OUCs and OUGs are the same network (up to direction).

c. The lower bound argument for shifting networks (Pippenger and Valiant 1976) shows that OHCs have to be of size O(NlogN) as well, for example by considering an N-HC as an (N/2,N)-shifting network.



The line separates networks with NlogN lower bound from networks with linear upper bound.

Figure 2.8 The relations among the networks.

Theorem 2.8 (Thompson (1978), Ofman (1967))

a) If A is an N-HC and B is an N-HG then D=A·B is an N-SG.

b) If D is an N-SG and C is an N-PN then D·C is an N-GPN.

c) If A is an N-HC, B is an N-HG and C is in N-PN, then A·B·C is an N-GPN.

Proof

For each GPN requirement, network A concentrates the r active input nodes to the first r output nodes, which are the first r input nodes of B. Network B produces the required copies of each input, and network C permutes them to the required positions.

Corollary 2.8

$GPN^{-}S(N) < 11.4N\log_3 N+o(N\log N)$
$GPN^{-}S(N) < 7.2N\log_2 N+o(N\log N)$.

Proof

Combining the results of corollaries 2.1, 2.3 and 2.5 and the results of chapter 5 cited in formula 2.8.

$GPN^{-}S(N) \leq 6N\log_3 N+3N\log_3 N+1.5N\log_2 N+o(N\log N) \leq$
$11.4N\log_3 N+o(N\log N)=$
$7.2N\log_2 N+o(N\log N)$.

2.7 Partitioners

A partitioner is a network which can implement the partitioning of resources into disjoint subsets, interconnected by disjoint subnetworks.

It is clear that any GPN can serve as a partitioner (Thompson (1978)), however a less powerful network is sufficient. (Masson, Gingher, Nakamura (1979))

Theorem 2.9

If A is an N-Generalizer, and B is an N-PN, then A·B is a partitioner for the output nodes of B.

Proof

By the definition of a generalizer, for each set of $k_1,k_2,\ldots k_r$ numbers $\sum k_j \leq N$, there is some set of input nodes $i_1,i_2,\ldots i_r$ such that i_j is connected to k_j output nodes. Let k_j be the number of terminals in the j'th subsystem. Subnetwork A establishes connection among k_j arbitrary output nodes of A, which are input nodes of B. Subnetwork B can establish connections from those k_j input nodes to the k_j output nodes which are to be connected.

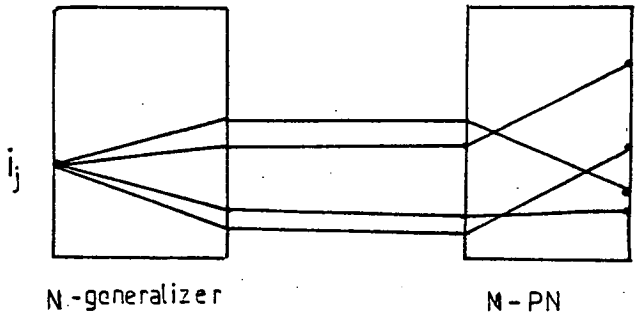


Figure 2.9 A Partitioner constructed from Generalizer and PN.

It is easy to construct an O(N) N-Orderpreserving Generalizer (OG). (We don't claim anything regarding the more interesting generalizer with different number of input and output nodes.) For example the network in figure 2.10 is an N-OG with 2(N-1) edges. If the given numbers are k_1, k_2, $\ldots k_r$, let input node i_j, connected to k_j output nodes, be the
$(\sum_{s<j} k_s)$ input node. It can be connected to
output nodes $(\sum_{s<j} k_s)$ up to $\sum_{k\leq j} k_s - 1$
The disadvantage of the network is its depth: N.

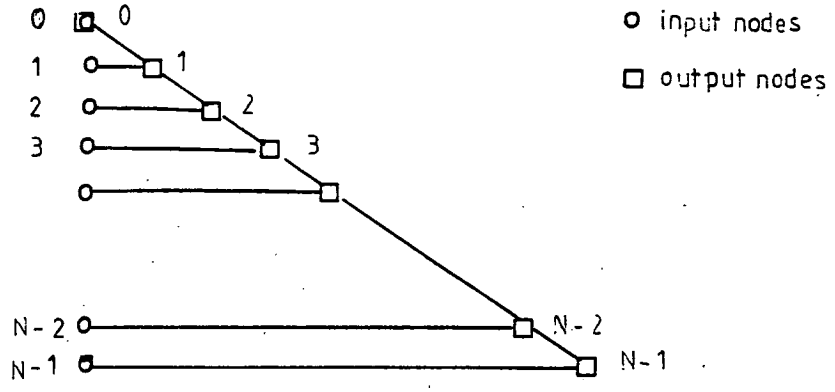


Figure 2.10 An N-OG

An improved construction can be obtained by adapting a construction of Ladner and Fischer (1977):

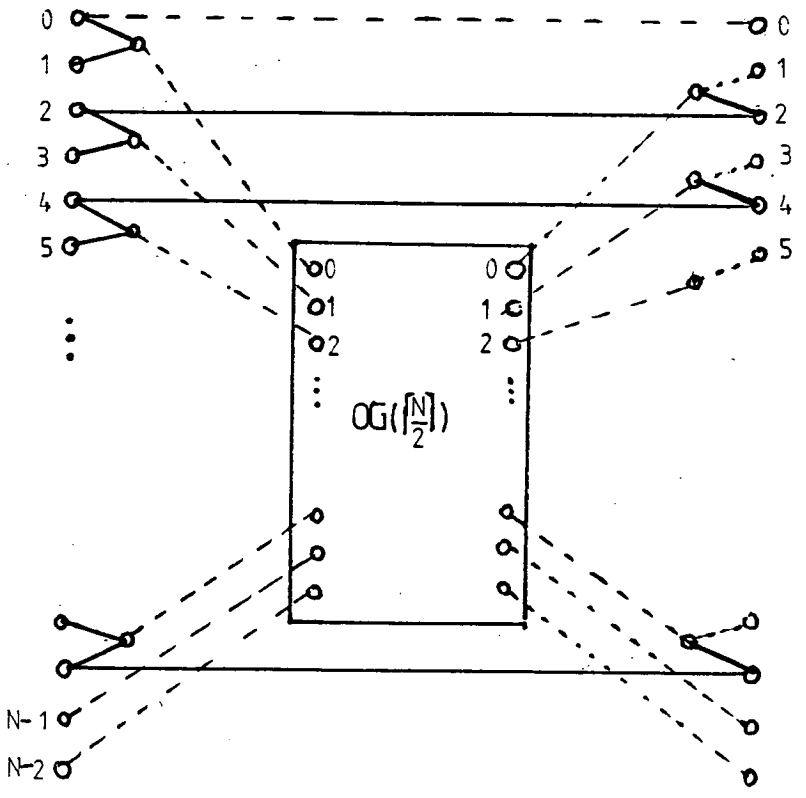
Theorem 2.10

The network in figure 2.11 is an N-OG with O(N) edges and O(logN) depth.

Proof

OG(N) is depicted in figure 2.11. It is built recursively from $OG(\lceil N/2 \rceil)$.

As in the previous network, the j'th active input node is the $(\sum_{s<j} k_s)$ input node.
It is connected to output nodes
$(\sum_{s<j} k_s)$ through $\sum_{s\leq j} k_s - 1$.



The full lines represent switches
The dashed lines denote identification of nodes.

Figure 2.11 The construction of OG(N)

The correctness proof for the construction of OG(N) from OG($\lceil N/2 \rceil$) is done by induction. If the index of an input node i_j is even it is connected directly to output node i_j. If the index of i_j is even and $k_j>1$ or if i_j is odd, it is connected to input node $\lfloor i_j/2 \rfloor$ of OG($\lceil N/2 \rceil$), which is an OG with the required properties, by the induction hypothesis, so it has paths to its output nodes $\lfloor i_j/2 \rfloor$ up to $\lfloor (i_j+k_j)/2 \rfloor -1$, from which there are paths to output nodes $\lfloor i_j/2 \rfloor \cdot 2$ up to i_j+k_j-1 of OG. which can be easily proved by induction.

It is easy to prove by induction that
$OG^{-}S(N) < 4N$.
Therefore the network is an OG with depth $2\lceil \log_2 N \rceil$ and size 4N.

Corollary 2.10

It is possible to build a partitioner of size $6N\log_3 N+o(N\log N)$ and depth $2(\log_3 N+\log_2 N)$.

Proof

The corollary derives from theorem 2.9 together with theorems 2.1 and 2.10.

3 LOWER BOUNDS FOR SUPERCONCENTRATORS

3.1 Introduction and layout of proof

In this chapter we give a lower bound of $4N-O((N \log N)^{1/2})$ for the number of computed nodes in a superconcentrator with binary indegree, and a lower bound of $5N-O(\log N)$ for the number of arcs in a superconcentrator with arbitrary indegree. Thus when a computation graph is a superconcentrator, it must have at least $4N-o(N)$ binary operations, and when it represents a communication network it must have at least $5N-o(N)$ switches. As mentioned in the introduction, this result gives a lower bound of $4N-o(N)$ additions for computing convolution and for computing the Discrete Fourier-Transform of prime order. The results in this chapter were obtained jointly with L.G. Valiant.

An (M,N) SC is a SC with M input nodes, N output nodes and capacity C=min(M,N). The minimal number of computed nodes in an (M,N) SC with binary indegree is denoted $SC^{-}N(M,N)$ and the minimal number of edges in an (M,N) SC with arbitrary indegree is denoted $SC^{-}S(M,N)$.

First, the problem of bounding below the number of nodes (or edges) in an (N,N) SC is reduced to the problem of bounding below the number of nodes (or edges) in an (M,N) SC by showing that

$$SC^{-}N(M,N) \geq SC^{-}N(M-1,N)+2$$

and applying this reduction N-M times to get

$$SC^{-}N(N,N) \geq SC^{-}N(M,N)+2(N-M).$$

Similarly for edges:

$$SC^{-}S(M,N) \geq SC^{-}S(M-1,N)+2$$

and

$$SC^{-}S(N,N) \geq SC^{-}S(M,N)+2(N-M).$$ (section 3.2)

In this reduction we use an argument similar to the argument of Schnorr (1974), as cited in Paterson (1976).

In section 3.3 it is shown that if the output nodes have indegree 2 then the number of internal computed

nodes in an (M,N) SC is bounded below by $N-4(N/M)\log N-2(N/M)$. Choosing M to be $(N \log N)^{1/2}$ yields in the binary indegree SC

$$SC^{-}N(N,N) \geq 4N-O(N \log N)^{1/2}$$

and for an arbitrary SC

$SC^{-}S(M,N) \geq 3N-o(3N)$, $SC^{-}S(N,N) \geq 5N-o(N)$. (section 3.4)

(All the logarithms in this chapter are to the base 2)

3.2 Bounding an N-SC by an (M,N)-SC

We can assume without loss of generality that there are no parallel edges in the SC and that each computed node has indegree greater than 1; a computed node with indegree 1 can be identified with its predecessor.

Lemma 3.1

In a directed acyclic (M,N)SC $SC^{-}S(M,N) \geq SC^{-}S(M-1,N)+2$.
If the SC has fanin 2 then $SC^{-}N(M,N) \geq SC^{-}N(M-1,N)+2$

Proof

The graph is acyclic and without parallel edges, therefore there is at least one computed node with two arcs incoming from two different input nodes. These two input nodes have node disjoint paths to any two output nodes, so at least one of them has a second outgoing arc to a computed node. (see figure 3.1)

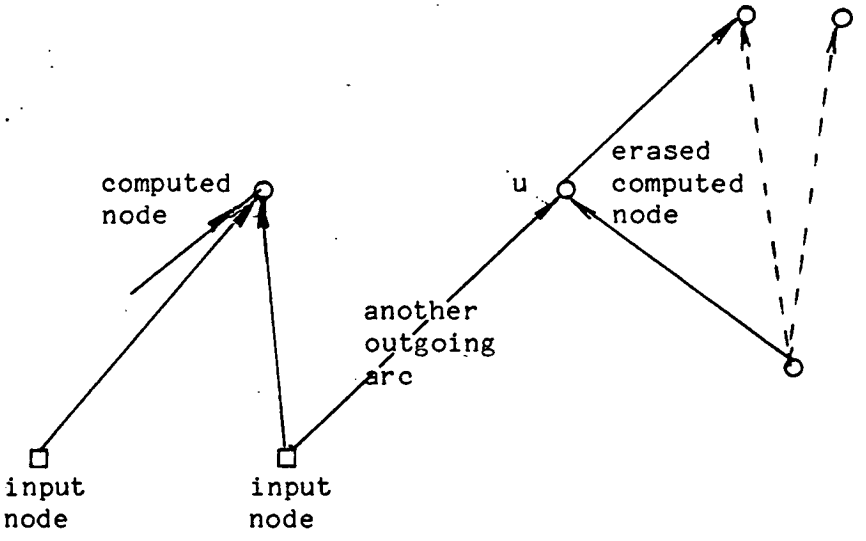


Figure 3.1

Erase from the SC this input node and at least two arcs which emanate from it, to get, if the resulting graph is a SC:
$SC^{-}S(M,N) \geq SC^{-}S(M-1,N)+2$.

If an erased arc was connected to a computed node with indegree 2, say u, erase u as well and connect directly the other (only one) predecessor of u to the successors of u. The indegree of the successors is not changed, the number of arcs is not increased, and creation of computed nodes with indegree 1 is prevented by this deletion.

When the graph has indegree 2, both outgoing arcs are directed to computed nodes with fanin 2. These nodes are erased from the SC to get, if the resulting graph is a SC:
$SC^{-}N(M,N) \geq SC^{-}N(M-1,N)+2$

The resulting graph is a SC.
In the original graph for each choice of $R \leq \min(M-1,N)$ input nodes, not including the erased one, there are R node disjoint paths from input to output. By skipping the erased computed nodes, the paths in the reduced graph are node disjoint.

If one of the erased computed nodes, say u, is an output node, its second predecessor, say w, can serve as an output node in the reduced graph; all the paths to u which are not from the erased input node have to pass through w, and they are node-disjoint from the paths to other output nodes. If w is erased too, its predecessor will serve as the new output node. (see figure 3.2)

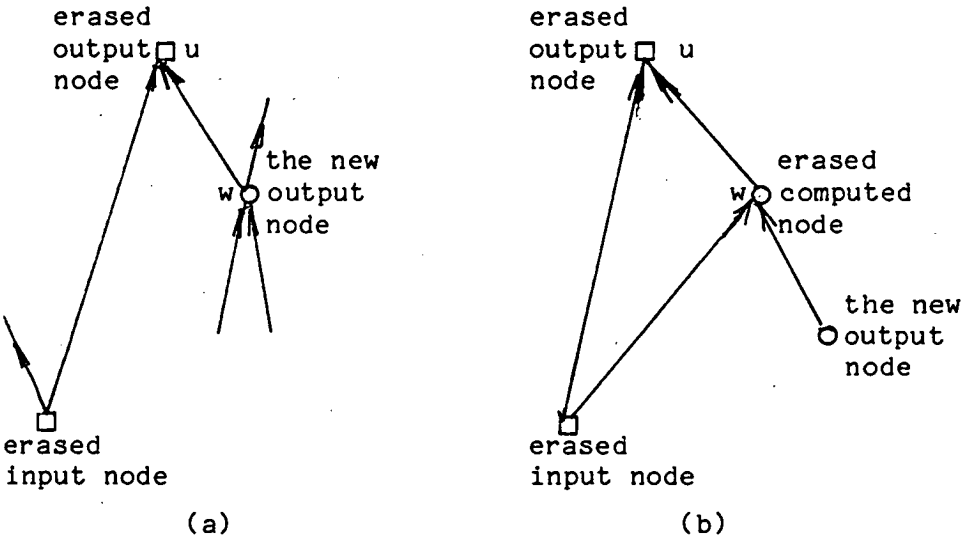


Figure 3.2

3.3 A lower bound for (M,N)-SCs

Theorem 3.1

If all the output nodes of an (M,N) SC have fanin 2, then the SC has more than $2N-4(N/M)\log N-2N/(M-1)$ computed nodes.

Construction

Let H be an (M,N) SC whose output nodes have fanin 2. Consider the subgraph H' which consists of all the output nodes and all the incoming arcs to the output nodes together with the nodes which are the tails of those arcs.

From H' construct an undirected graph G as follows: (see figure

3.3)

a) The edges of G correspond to the output nodes of H'.

b) The nodes of G correspond to the nodes with fanout greater than 0 in H': each internal computed node in H' is mapped to a regular node in G and each input or output node with outdegree greater than 0 in H' is mapped to a special node in G. The number of edges in G equals N, the number of output nodes in H.

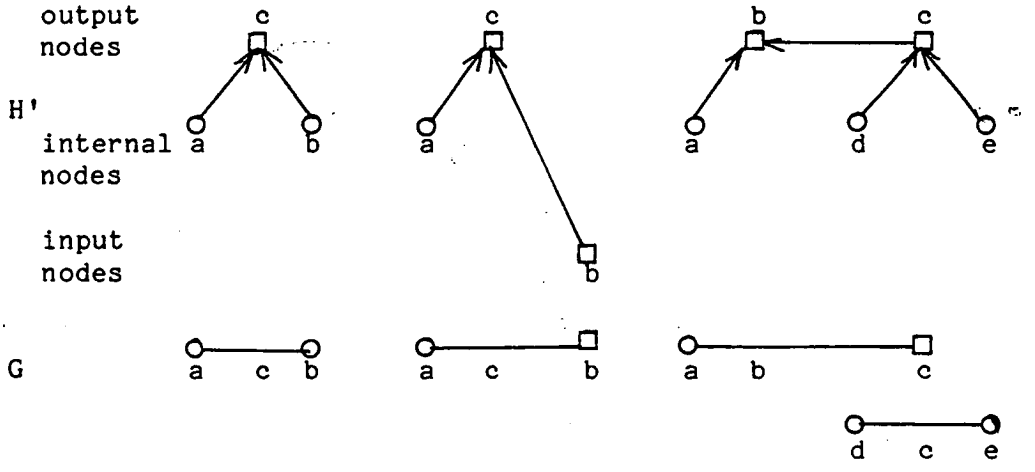


Figure 3.3 The mapping of H' to G

Note that an output node with fanout > 0 in H' is represented by a special node and by a separate edge in G.

Proof outline

Proving that the number of edges in G can exceed the number of regular nodes by at most $4(N/M)\log N+2N/(M-2)$ will show that the number of output nodes in H can exceed the number of internal computed nodes by at most that much, and thus, that the number of internal computed nodes plus output computed nodes is as claimed.

Lemma 3.2

Any connected subgraph of G which contain 2 special nodes must contains at least M-1 edges.

Any connected subgraph of G which contains 2 closed cycles must contain at least M+1 edges. In this pages the term "closed cycle" means what is normally called "cycle" or "closed "path".

Any connected component of G which contains one closed cycle and one special node contains at least M edges.

Proof

Denote by a,b,c

respectively	the ocurrence in G of:
a)	a special node induced by an input node of H'
b)	a special node induced by an output node of H'
c)	a closed cycle

Case a,a There are 2 special nodes induced by input nodes in a connected subgraph of G with j edges.

In the corresponding subgraph of H' there are j output nodes which can be connected to M-2 input nodes via at most j-1 internal nodes (see figure 3.4).
If M-2 > j-1 then there are no node disjoint paths from some j input nodes to those output nodes, contradicting the definition of a SC. Therefore M-2 $\leq$ j-1,
j $\geq$ M-1.

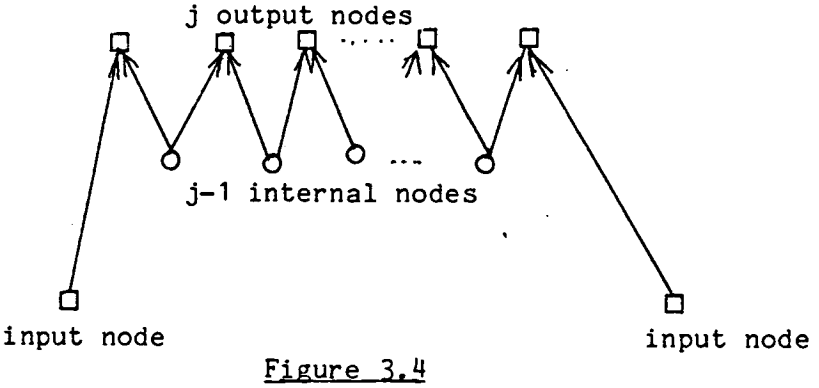


Figure 3.4

Case a,b In a connected subgraph of G with j edges there is one special node induced by an input node and one special node induced by an output node of H'. Then in H', there are j node disjoint paths to the j+1 corresponding output nodes (including the additional one) from M-1 input nodes (see figure 3.5). If M-1 > j, then for some j+1 input nodes there are no node disjoint paths to those j+1 output nodes, and therefore
$j \geq M-1$.

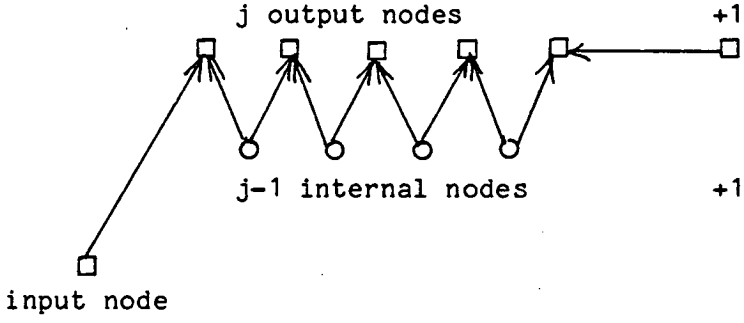


Figure 3.5

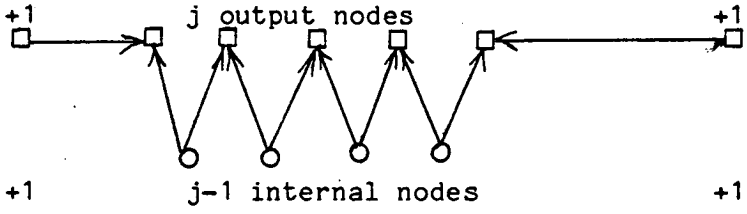


Figure 3.6

Case b,b There is a connected subgraph in G with 2 special nodes induced by output nodes of H' (see figure 3.6). Then in the corresponding subgraph of H' there are j+2 output nodes connected to the input via at most j+1 internal and output nodes. If j+1 < M, then there are no node disjoint paths from any j+2 input nodes to those j+2 output nodes. Therefore $j+1 \geq M$,
$j \geq M-1$.

Case c,c There are 2 closed cycles in a connected subgraph of G with j edges. Then in H' there are 2 closed cycles in a connected subgraph with j output nodes (see figure 3.7). Those j output nodes are connected to the input via j-1 (at most) internal nodes. If $j-1 < M$, then for some j input nodes there are no node disjoint paths to those output nodes. Therefore $j-1 \geq M$,
$j \geq M+1$.

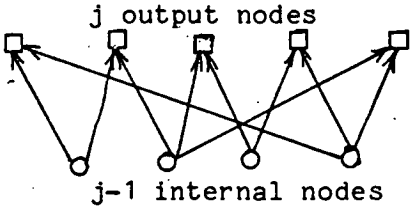


Figure 3.7

Case a,c There is a connected subgraph of G with j edges containing a closed cycle and a special node induced by an input node of H'. Then in H' there are j output nodes connected to M-1 of the input nodes via (at most) j-1 internal nodes (see figure 3.8). If $M-1 > j-1$ then there are no node disjoint paths from some j input nodes to those j output nodes. Therefore $j-1 \geq M-1$,
$j \geq M$.

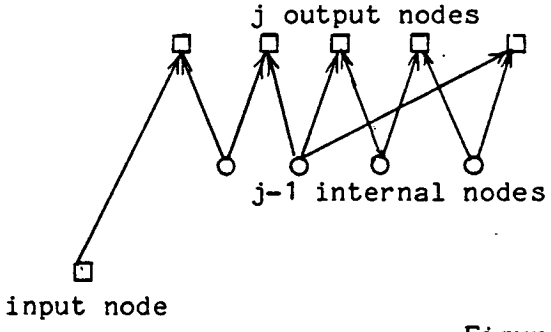


Figure 3.8

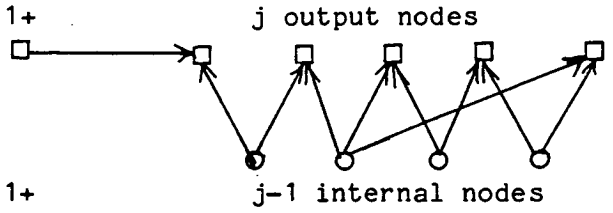


Figure 3.9

Case b,c There is a subgraph of G with j+1 edges which contains a closed cycle and a special node induced by an output node of H'. Then in H' there are j+1 output nodes connected to the input via at most j nodes (see figure 3.9). If $j < M$, then for j+1 input nodes there are no node disjoint paths from input to output. Therefore $j \geq M$.

Lemma 3.3

In any connected component of G with q edges which satisfies

$$k\,\frac{M-1}{2} \leq q < (k+1)\,\frac{M-1}{2} \quad ,$$

there are at most k special nodes. If k=0 or 1, there is at most one special node.

Proof

Assign to each special node all the edges within distance (M-1)/2 from it. If there are more than k special nodes, at least (k+1)(M-1)/2 edges get assigned in this way. There are less than (k+1)(M-1)/2 edges in the component, so that at least one edge is assigned to 2 different nodes. Those 2 nodes and the edges assigned to them on the path connecting them, are in a connected subgraph which contains at most M-2 edges, contradicting lemma 3.2.
The second assertion is proved in lemma 3.2.

Lemma 3.4

In any graph G which contains E edges and E-D nodes there exists a subgraph G' with $E' < 4(E/D) \log E$ edges which contains two cycles, i.e., there is a subgraph with $E' < 4(E/D) \log E$ edges and with $N' < E'$ nodes.

Proof

Let T=E/D.

Perform the following transformation on G:

a) Delete T+1 edges from any "degree 2 chain" of length greater than T edges and delete any isolated node which was formed by the deletion.

b) Cut off every edge of which one end has degree 1 (as these cannot participate in any cycle) and delete every isolated node.

Perform those transformations as much as possible, (see figure 3.10), to produce a graph G' with

1) No nodes with degree 1.

2) No degree-2 chains whose length is greater than T.

Assume that the first transformation was performed k times, and the second one was performed j times. Each application of transformation a) eliminates T+1 edges. If $k \geq D$, then at least $D((E/D)+1)$ edges are deleted which is impossible. Hence clearly:

3) $k < D$.

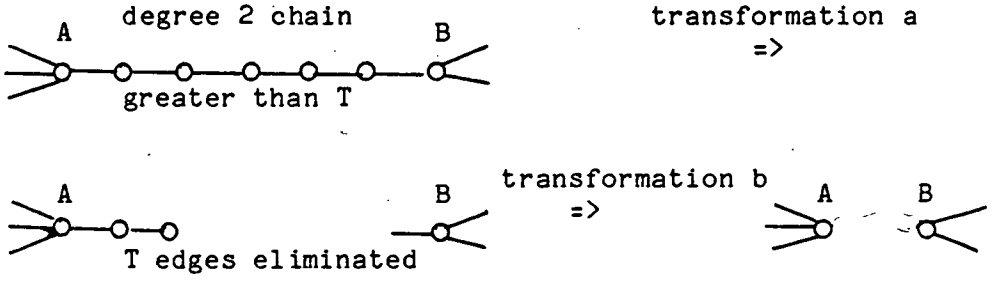


Figure 3.10

Each application of transformation a) reduces the number of edges by T+1 and the number of nodes by at least T. Each application of transformation b) reduces the number of edges and nodes by 1. G' has $E'=E-k(T+1)-j$ edges and $N' \leq E-D-kT-j$ nodes. $k < D \Rightarrow N' < E'$ and therefore G' contains a connected subgraph with more edges than nodes.
The proof proceeds by showing that there is such a subgraph of size less than $4(E/D)\log D$.

Let $d(V_i,V_j)$, the <u>distance</u> between V_i and V_j, be the length of a minimal path from V_i to V_j. For any two cycles C_1, C_2 in G', let G_{12} be the minimal connected subgraph which contains them. If C_1 and C_2 are in the same connected component of G', then $\underline{d}_{12}$, <u>the diameter</u> of G_{12} is the maximal distance between any two nodes in the subgraph; $d_{12} = \infty$ otherwise.

Choose C_1 and C_2 so that for any two cycles C_3, C_4, $d_{12} \leq d_{34}$. Let V_0 be a node which satisfies:

$$\exists\, V_1 \in C_1 \quad d(V_0,V_1)=\lceil d_{12}/2 \rceil \text{ and } \exists\, V_2 \in C_2 \quad d(V_0,V_2)=\lfloor d_{12}/2 \rfloor .$$

(V_0 is a special centre of G_{12}, as defined in Ore, pp 27-29)

Construct a breadth-first search tree starting from V_0. By the choice of V_0, C_1 and C_2 will be closed either at the same distance from the root, or C_1 will be closed at a distance longer by one than C_2. No other cycle can be closed at a distance shorter than C_1 from the root, for otherwise such a cycle and C_2 would be contained in a subgraph with diameter shorter than d_{12}.

By transformation b) G' does not contain nodes with degree 1, so that each branching in the tree can stop only by closing a cycle.

By transformation a), after at most T edges a branching must occur in each branch of the tree. The number of tree edges,

constructed up to height $T \cdot x$ where C_1 closes, is at least $T \cdot 2^x$. This number is less than E.

$T \cdot 2^x < E, \quad x < \log_2(E/T) = \log_2 D.$

The height of the tree when C_1 closes is less than T log D, and the number of edges in G_{12} is less than $4(E/D) \log D < 4(E/D) \log E$

Proof of Theorem 3.1

The proof that the number of edges in G exceeds the number of regular nodes in G by at most $4(E/M)\log E+2E/(M-2)$ is done by induction on the number of connected components.

Assume that G has one connected component G' with E' edges and the other connected components with E-E' edges. If $E' < M-1$, then by lemma 3.2 G' can contain at most one cycle or one special node and the number of regular nodes of G', R(G'), is at least the number of edges E'.

If $E' \geq M-1$, then by lemma 3.3 G' contains at most $2E'/(M-1)$ special nodes. By lemma 3.2 there cannot be two cycles within M edges, so as a corollary from lemma 3.4 the difference D' between the number of nodes and edges has to satisfy the equation

$$4(E'/D')\log E' > M$$

which is

$$D' < (4E' \log E')/ M.$$

Therefore

$$N' = E'-D' > E'-(4E'\log E')/M,$$

and hence

$$R(G') > E' - 4(E'\log E')/M - 2E'/(M-1).$$

If G has more than one connected component, using the induction hypothesis

$$R(G-G') > (E-E') - 4((E-E')\log(E-E'))/M - 2(E-E')/(M-1)$$

adds to

$$R(G) > E - 4(E \log E)/M - 2E/(M-1)$$

The number of edges of G equals N, The number of output nodes of H, and each regular node in G corresponds to an internal computed node in H. Thus the number of internal computed nodes in G is at least

$$N-4(N \log N)/M - 2N/(M-1).$$

By counting the output nodes also, we get that the number of computed nodes is greater than

$$2N - 4(N \log N)/M - 2N/(M-1)$$

3.4 Conclusion: lower bounds for SCs

Theorem 3.2

The number of computed nodes in a directed acyclic N SC with fanin 2 is at least
$4N-O(N\log N)^{1/2}$

Proof

Applying the reduction described in lemma 3.1,

$SC^{-}N(M,N) \geq SC^{-}N(M-1,N)+2$ N-M times gives

$SC^{-}N(N,N) \geq SC^{-}N(M,N)+2(N-M)$. Applying theorem 3.1 yields

$SC^{-}N(N,N) > 2N-4(N/M)\log N-2N/(M-1)+2(N-M)$.

Choosing $M=\Omega N\log N)^{1/2}$ results in

$SC^{-}N(N,N) > 4N-(32N\log N)^{1/2}-2N/((2N\log N)^{1/2}-1)$

Theorem 3.3

The number of edges in a directed acyclic N SC with arbitrary indegree is at least $5N-O(\log_2 N)$

Proof

Apply the reduction described in lemma 3.1 N-M times to get $SC^{-}S(N,N) \geq SC^{-}S(M,N)+2(N-M)$. Erase from the resulting (M,N) SC any output node which has indegree 3 or more. Suppose there are K such output nodes, then $SC^{-}S(M,N) \geq SC^{-}S(M,N-K)+3K$. The resulting (M,N-K) SC has all of its output nodes with fanin 2. By theorem 3.1 it has at least
$2(N-K)-4((N-K)/M)\log(N-K)-2(N-K)/(M-1)$ computed nodes. Each such node has at least 2 incoming arcs and the number of the arcs in the graph is at least
$2N-2M+3K+4(N-K) - 8((N-K)/M)\log(N-K)-4(N-K)/(M-1) =$

$5N-2M + (N-K)\{1-8(\log_2(N-K)/M)-4/(M-1)\}$

Choose $M=16\log_2(N)$ to get

$SC^{-}S(N) > 5N-32\log_2(N)$

4 EXISTENTIAL UPPER BOUND FOR SUPERCONCENTRATORS

4.1 Introduction

In this chapter the existential upper bound on the size of SCs which was derived by Pippenger (1977a) is improved from 39N+O(logN) edges to 38.5N+O(logN) and from 40N edges to 39.05N edges. The proof method follows Pippenger's construction. The main difference is that in seeking better numerical values the construction was done using parameters, which were optimized only at the end. In producing the 39.05 upper bound, results from chapters 2 and 5 are applied, i.e. the existance of constructions for SCs of size $3N\log_2 N-(5/4)N$ edges. The 38.5N result has been announced recently also by Chung (see Gabber and Galil(1979)). When the indegree of each computed node is restricted to 2, the old construction yields a SC with 36N+O(log N) computed nodes, and the new one results in a SC with 32.5N+O(logN) computed nodes.

The proof that SCs with 39N+O(logN) edges exist is based on the proof that Concentrators with certain properties exist.
An (N,M,C) Concentrator is a graph with N input nodes, M output nodes and capacity C; for every $K \leq C$ and every choice of K input nodes there are K node disjoint paths from those input nodes to some K output nodes.

4.2 A family of concentrators

Construction

Let G belong to a family of bipartite graphs, each graph in the family has N input nodes and aN output nodes (a<1). Let the indegree of each output node be b (an integer), and the outdegree of each input node ab (an

integer). Each graph has E=abN edges. Index the nodes and edges from 0 to N, to aN and to abN respectively. Let H be a permutation on {1,...,E}. For each permutation construct a graph in the family as follows:
The i'th arc is directed from input node i (mod N) to output node Hi (mod aN). Assign equal probability to each graph in the family.

Claim 4.1

The probability of choosing a graph in the family which is not an $(N,aN,\lfloor cN \rfloor)$ concentrator (i.e. is a "bad graph") is limited above by:

[i] $$\sum_{K=2}^{cN} \frac{\binom{N}{K}\binom{aN}{K}\binom{bK}{abK}}{\binom{abN}{abK}} <$$

[ii] $$\sum_{K=2}^{cN} \frac{\binom{bK}{abK}}{\binom{(ab-1-a)N}{(ab-2)K}}$$

which is greater than 1 for a,b,c,d=N/⌊Nc⌋ which satisfy:

[1]= $$\frac{(da)^{da}\,(d-1)^{(d-1)(ab-1)}}{(da-1)^{da-1}\,(1-a)^{b-ab}\,a^{ab}\,d^{d(ab-1)}} > 1$$ (N large enough)

and less than 1 if

[2] $b \geq 2/(2a-1)$ and

[3] $ab \geq 3$ and

[4] $$\frac{(ab-2)^{ab-2}\,(d(ab-1-a)-(ab-2))^{d(ab-1-a)-(ab-2)}}{a^{ab}\,(1-a)^{b(1-a)}\,(d(ab-1-a))^{d(ab-1-a)}} < 1$$

(for large enough N)

{note: Inequalities [2]-[4] will provide criterion for recognizing "good" graphs. Inequality [1] is an easy way to eliminate graphs which are not recognizable as "good" by approximation [i]. We shall see that all the graphs (with smaller number of edges or nodes than in the construction of Pippenger), that are not

recognized as "bad" by [1], are recognized as "good" by [2]-[4], and therefore approximation [ii] gives the same results as approximation [i].}

Proof

A graph in the family is a concentrator if for each $K \leq C$ there are K disjoint connections from input to ouput, i.e. there is a K matching in the bipartite graph. By Hall's theorem there is a K matching if there is no set of K input nodes which is connected to less than K output nodes.

Let A be a set of K input nodes and B a set of K(*) output nodes. A corresponds to a set of abK (outgoing) arcs and B corresponds to a set of bK (incoming) arcs. For fixed K,A,B there are $[bK]_{abK}(abN-abK)!$ ways of choosing a permutation so that A will be directed into B. (The first term is the number of possibilities to direct all the arcs from A into B, and the second term is the number of permutations of the remaining abN-abK arcs).

There are $\binom{N}{K}$ ways to choose A and $\binom{aN}{K}$ ways to choose B.

There are altogether (abN)! permutations.

The probability of choosing a graph in which for some K there is no K flow is limited above by:

$$\sum_{K=2}^{cN} \frac{\binom{N}{K}\binom{aN}{K}\,[bK]_{abK}\,(abN-abK)!}{(abN)!} =$$

$$\sum_{K=2}^{cN} \frac{\binom{N}{K}\binom{aN}{K}\,\frac{(bK)!}{(bK-abK)!}\,\frac{(abK)!}{(abK)!}\,(abN-abK)!}{(abN)!} =$$

$$=\sum_{K=2}^{cN} \frac{\binom{N}{K}\binom{aN}{K}\binom{bK}{abK}}{\binom{abN}{abK}} = [i]$$

Since

(*) K-1 would have been sufficient here.

$$\binom{abN}{abK} \geq \binom{N}{K}\binom{aN}{K}\binom{(ab-a-1)N}{(ab-2)K}$$

if $ab \geq 2$ the probability is less than

$$[ii] \quad \sum_{K=2}^{cN} \frac{\binom{bK}{abK}}{\binom{(ab-1-a)N}{(ab-2)K}}$$

Lemma 4.1

If

$$[1] = \frac{(da)^{da}\,(d-1)^{(d-1)(ab-1)}}{(da-1)^{da-1}\,(1-a)^{b-ab}\,a^{ab}\,d^{d(ab-1)}} > 1$$

is not satisfied, than the sum [i] exceeds 1 for large enough N.

Proof

Consider the last addend in the sum [i].

(In the following cN stands for $\lfloor cN \rfloor$, $d = N/\lfloor cN \rfloor$)

$$\frac{\binom{N}{cN}\binom{aN}{cN}\binom{bcN}{abcN}}{\binom{abN}{abcN}} =$$

$$= \frac{N!\,(aN)!\,(bcN)!\,(abN-abcN)!}{(cN)!\,(N-cN)!\,(cN)!\,(aN-cN)!\,((b-ab)cN)!\,(abN)!}$$

Using Stirling's formula together with the fact that $e^x < 1/(1-x)$ we have the following limits on N!:

$(2\pi N)^{1/2}(N/e)^N < N! < (2\pi N)^{1/2}(N/e)^N(12N)/(12N-1)$

Using these limits, all the powers of $(\lfloor cN \rfloor/e)$ sum to 0, and the sum is bounded above by

$$\left(\frac{d^d\,(da)^{da}\,b^b\,((d-1)ab)^{(d-1)ab}}{(d-1)^{d-1}\,(da-1)^{da-1}\,(b(1-a))^{b(1-a)}\,(dab)^{dab}}\right)^{cN}$$

$$* \; (1/(2\pi cN))\,(aN/((aN-cN)(1-a)))^{1/2}$$

$$* \; \frac{12N}{12N-1}\,\frac{12aN}{12aN-1}\,\frac{12bcN}{12bcN-1}\,\frac{12(d-1)abcN}{12(d-1)abcN-1}$$

For N big enough the leading term is the first one, and if this

term is bigger than 1, then [i] exceeds it as well. We look for:

$$[1]=\frac{(da)^{da}\,(d-1)^{(d-1)(ab-1)}}{(da-1)^{da-1}\,(1-a)^{b-ab}\,a^{ab}\,d^{d(ab-1)}}>1$$

Eventually we will be interested only in c=1/2. For c=1/2 and N big enough, d->2 and [1] reduces to:

$$[1']\quad\frac{(2a)^{2a}}{(2a-1)^{2a-1}\,(1-a)^{b-ab}\,a^{ab}\,2^{2ab-2}}>1$$

Using the approximation (ii) we will check only those parameters that violate [1'].

Lemma 4.2

Let each term in [ii] be called L_K. For $b\geq 2/(2a-1)$ L_K/L_{K+1} is an increasing function of K and the biggest term in the sum [ii] is either the first or the last one.

Proof

$$\frac{L_{K+1}}{L_K}=\frac{[b(K+1)]_b\;[(ab-2)(K+1)]_{ab-2}}{[ab(k+1)]_{ab}[(b-ab)(K+1)]_{b-ab}[(ab-1-a)N-(ab-2)K]_{ab-2}}=$$

$$=\frac{[bK+b]_{b-ab}\,[bK+ab]_{ab}\,[(ab-2)K+(ab-2)]_{2ab-2-b}}{[abK+ab]_{ab}}$$

$$*\;\frac{[(ab-2)K+(b-ab)]_{b-ab}}{[(b-ab)K+(b-ab)]_{b-ab}\,[(ab-1-a)N-(ab-2)K]_{ab-2}}$$

If 2ab-2-b>0 each term is a non decreasing function of K, and at least the first one is an increasing function of K, so the expression is an increasing function of K.

Suppose that there is a maximal element L_{K+1} which is not the first or the last in the sum. Then $L_{K+1}>L_K$ and $L_{K+1}>L_{K+2}$ implies
$(L_{K+1})/L_K > (L_{K+2})/(L_{K+1})$
contradicting the above result.

Lemma 4.3

If the biggest element in the sum [ii] is the first element, then for big enough N the sum is less than 1 when [3] is satisfied.

Proof

If the first element is the biggest element in the sum then

$$\sum_{K=2}^{cN} \frac{\binom{bK}{abK}}{\binom{(ab-1-a)N}{(ab-2)K}} < cN \frac{\binom{2b}{2ab}}{\binom{(ab-1-a)N}{2ab-4}} =$$

$$= \frac{cN \quad [2b]_{2ab}}{[2ab]_4 [(ab-1-a)N]_{2ab-4}}$$

If $ab \geq 3$, there is N in the numerator and $O(N^{2ab-4})$ in the denominator, so that for N big enough the sum is less than 1, and the probability that the graph is bad is less than 1.

Lemma 4.4

If the biggest element in the sum [ii] is the last element, then for N big enough the sum is less than 1 when [4] is satisfied.

Proof

If the biggest element in the sum is the last element then (in the following cN stands for $\lfloor cN \rfloor$, $d=N/\lfloor cN \rfloor$)

$$\sum_{K=1}^{cN} \frac{\binom{bK}{abK}}{\binom{(ab-1-a)N}{(ab-2)K}} < cN \frac{\binom{bcN}{abcN}}{\binom{(ab-1-a)N}{(ab-2)cN}} =$$

$$= cN \frac{(bcN)!\ ((ab-2)cN)!\ ((ab-1-a)N-(ab-2)cN)!}{(abcN)!\ (b(1-a)cN)!\ ((ab-1-a)N)!}$$

Using the limits:

$(2\pi N)^{1/2}(N/e)^N < N! < (2\pi N)^{1/2}(N/e)^N(12N)/(12N-1)$

all the exponents of $(\lfloor cN \rfloor/e)$ sum to 0, and the term is bounded above by

$$cN \left(\frac{(ab-2)^{ab-2}\ (d(ab-1-a)-(ab-2))^{d(ab-1-a)-(ab-2)}}{a^{ab}\ (1-a)^{b(1-a)}\ (d(ab-1-a))^{d(ab-1-a)}} \right)^{cN}$$

$$* \left(\frac{(ab-2)\ ((ab-1-a)d-(ab-2))}{ab\ (1-a)\ d(ab-1-a)} \right)^{1/2}$$

$$* \quad \frac{12bcN}{12bcN-1} \frac{12(ab-2)cN}{12(ab-2)cN-1} \frac{12((ab-1-a)N-(ab-2)cN)}{12((ab-1-a)N-(ab-2)cN)-1}$$

For N big enough the leading term is the first one, and if this term is less than one, then the sum is less then 1; if

$$[4] = \frac{(ab-2)^{ab-2}\ (d(ab-1-a)-(ab-2))^{d(ab-1-a)-(ab-2)}}{a^{ab}\ (1-a)^{b(1-a)}\ (d(ab-1-a))^{d(ab-1-a)}} < 1$$

the probability for a "bad" graph is less than 1.

The four lemmas prove claim 4.1.

4.3 Construction of an "optimal" SC

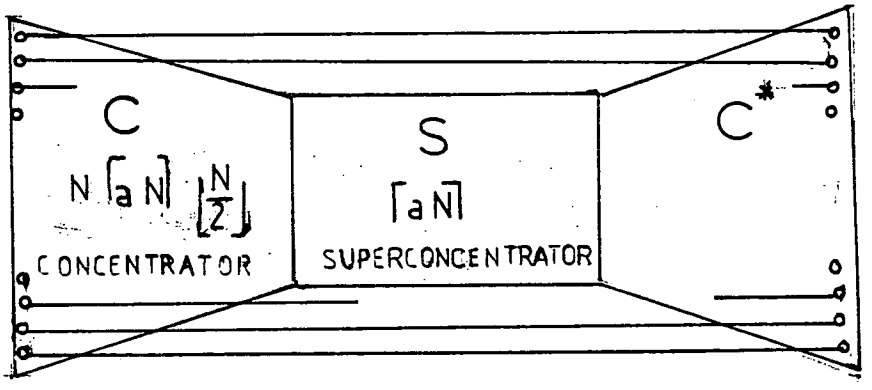
Construction of an N-SC

Let $N' = \lceil \lceil aN \rceil / a \rceil$, let C be an $(N', \lceil aN \rceil, \lfloor (1/2)N \rfloor)$-Concentrator and let S be an $\lceil aN \rceil$-SC built recursively. Construct the network $C \cdot S \cdot C^*$ and add in this network for each i an edge from input node i to output node i. Delete the last $N'-N$ input and output nodes

and the arcs incident with them.

Claim 4.2

The resulting network is an N-SC (see figure 4.1) which satisfies
$SC^{-}S(N)<(2ab+1)N+SC^{-}S(\lceil aN \rceil)$



Recursive construction of SuperConcentrator
Figure 4.1

Proof
Given any $K \leq \lfloor N/2 \rfloor$ superconcentration assignment, there are K active input nodes and K active output nodes. In C there are K node disjoint paths from those K active input nodes to some K output nodes, which are input nodes of S. Let those nodes be the active input nodes in S. In C^* there are K node disjoint paths from some K input nodes to the given K active output nodes in C^* (because C is a concentrator). Let those input nodes of C^* be the active output nodes of S. S is a SC by the inductive assumption, therefore it is possible to establish K node disjoint paths from its active input nodes to its active output nodes, thus establishing K node disjoint paths from input to output in $C \cdot S \cdot C^*$.

If $K > \lfloor N/2 \rfloor$, Let K_1 be the number of active input nodes which are connected by an arc to active output nodes, and K_0 the number of active input nodes (and active output nodes) which have no

direct connection to active output (input) node.
$2K_0+K_1<N$, and $K_0 \leq \lfloor N/2 \rfloor$, therefore it is possible to establish K_0 node disjiont paths from the remaining K_0 active input nodes to the remaining K_0 active output nodes in $C \cdot S \cdot C^*$.

The number of arcs in the resulting SC satisfies
$SC^-S(N) \leq (2ab+1)N+SC^-S(\lceil aN \rceil)$
which has a solution
$SC^-S(N) = (2ab+1)(1/(1-a))N+O(\log N)$.

Assume that the indegree of each node is limited to 2. Each node with indegree d can be represented by a tree with indegree 2 and d-1 nodes. The above construction gives
$SC^-N(N) \leq (b-1)\lceil aN \rceil+abN+SC^-N(\lceil aN \rceil)$
which has a solution
$SC^-N(N) \leq (2abN-aN) \cdot (1/(1-a))+O(\log N)$ We wish to combine claims 4.1 and 4.2 to yield SCs with fewer edges of nodes than the construction of Pippenger(1977a) (i.e. those that have $(2ab+1)/(1-a)<39$ or $(2ab-a)/(1-a)<36$). It can be verified that if none of the above is to be violated, then it is sufficient to consider the combinations summarized in the following table.

a	b	ab	SC_S	SC_N	success or failure
3/5	5	3	18		[1] satisfied
3/4	4	3	28		[1] satisfied
4/5	5	4	45	36	success
4/6	6	4	27		[1] satisfied
4/7	7	4	21		[1] satisfied
5/6	6	5	66		not checked
5/7	7	5	38.5	32.5	success
5/8	8	5	30		[1] satisfied
5/9	9	5	25		[1] satisfied
6/11	11	6	28.6		[1] satisfied
6/10	10	6	33		[1] satisfied
6/9	9	6	39	36	success (Pippenger)
7/11	11	7	41.25	36.75	[2] not satisfied
7/12	12	7	36		[1] satisfied
7/13	13	7	33		[1] satisfied
8/15	15	8	36.42		[1] satisfied
9/14	14	9	54		not checked
8/14	14	8	39.33	36	[1] satisfied

From the table we see that the minimal SC which can be found in this way satisfies a=5/7, b=7, $SC^{-}S(N)<38.5N+O(\log N)$, $SC^{-}N(N)<32.5N+O(\log N)$. All the candidates for SCs with fewer than 38.5N edges violate condition [1], and therefore this is the best SC which can be constructed with the approximation of the sum [i].

4.4 An exact upper bound for SCs

Claim 4.3

$SC-S(N)<39.05N$

Proof

Using the result from chapter 5:

$SC^{-}S(N)<3N\log_2 N-(5/4)N+2$

gives for $N<2^{13}$ $SC^{-}S(N)<38.5N$

For $N>2^{13}$ use the recursive construction:

$$SC^{-}S(N)<(2ab+1)N+SC^{-}S(\lceil aN \rceil)$$

with the best numerical values found, a=5/7 b=7.

Condition [4] is satisfied for all $N>2^{13}$, and

$SC^{-}S(N) < 11N + SC^{-}S(\lceil (5/7)N \rceil) < 11N+SC^{-}S(5\lceil N/7 \rceil)$

Let $F(N)=5\lceil N/7 \rceil$ $F^0(N)=N$; $F^{i+1}(N)=F(F^i(N))$

and let t satisfy: $F^t(N) > 2^{13} \geq F^{t+1}(N)$

$SC^{-}S(N) < 11N + SC^{-}S(F(N))$

applying the recursion t times :

$SC^{-}S(N)<11(F^0(N)+F^1(N)+\ldots F(^t(N)\} + F^{t+1}(N)$

$F(N)=5\lceil N/7 \rceil < 5\{(N/7)+(6/7)\}$

It is easy to check, using induction that for $i>0$ $F^i(N)<(5/7)^iN+15$

therefore

$SC^{-}S(N) <$

$11\{(5/7)^0N+((5/7)^1N+15)+((5/7)^2N+15+\ldots((5/7)^tN+15)\}+$

$+SC\_S(F^{t+1}(N))$

$< 11\ \frac{1-(5/7)^{t+1}}{1-(5/7)}\ N + 15t) + \frac{77}{2}((5/7)^{t+1}N+15) =$

$= 77/2\ N+ 11\cdot 15(t+7/2)$

$SC^{-}S(N) < 38.5N + 165(t+(7/2))$

For $N>2^{13}=8192$ $F(N)<5((N/7)+(6/7))=((5/7)+(30/7N))N<$

$((5/7)+(30/(7\cdot 8192))N$

This is true for $N'=N,F(N),F^2(N),\ldots F^t(N)$

By induction $F^t(N) < (40990/57344)^t\cdot N$

$2^{13}<F^t(N)$; $8192<(40990/57344)^tN$

$$t< \frac{\log \frac{N}{8192}}{\log \frac{57344}{40990}} = \frac{\log N - \log 8192}{\log 57344 - \log 40990}$$

For $N>8192$ $((\log N)/N) < ((\log 8192)/8192)$

$\log N < N \cdot ((\log 8192)/8192)$ and

$$t < \frac{\frac{N \log 8192}{8192} - \log 8192}{\log \frac{57344}{40990}} < \frac{13\cdot 2.065}{8192} N - 13\cdot 2.064$$

therefore $SC^{-}S(N) < 38.5N + 165\cdot\frac{13\cdot 2.065}{8192} N < 39.05\ N$

5. SMALL SIZE SUPERCONCENTRATORS

In this chapter lower and upper bounds are given for SCs of sizes up to 5. Some of the SCs constructed can serve as building blocks in the recursive construction discussed in chapter 2 and yield a lower multiplicative constant in that construction. In the second half of the chapter new upper bounds on the additive complexity of the DFT for sizes 3 and 5 are derived using particular SCs found in this chapter.

5.1 A classification of SCs

In order to discuss size bounds for SCs (and UCs), they are differentiated according to their use and structure. In particular, when building SCs (or HCs) recursively it is possible to relax some of the restrictions imposed on the network, if the resulting network can be guaranteed to have the required properties.

Claim

If in the recursive construction of networks:

a) An output node with outdegree > 0 is identified with an input node whose indegree = 0,

b) an output node with outdegree = 0 is identified with an input node whose indegree > 0,

then the resulting network remains acyclic and retains its routing properties as in the case when output nodes with outdegree = 0 are identified with input nodes with indegree = 0. (The indegree might be changed).

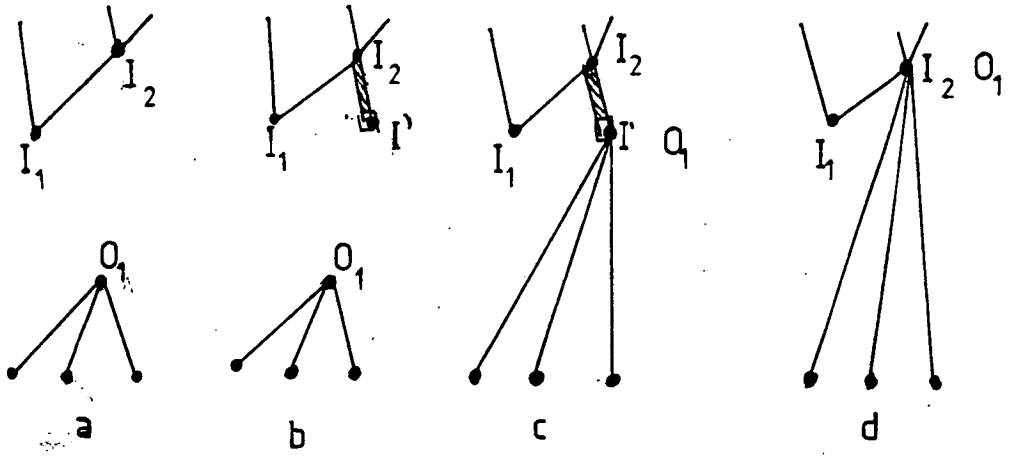
Proof

a) For each output node with outdegree > 0 add an additional node following it and let the added node be the new output node. The new output node has outdegree = 0. Identify it with an input node whose indegree = 0.

b) For each input node whose indegree > 0 add an additional node preceding this input node and let it be the new input node. The new input node has indegree=0. Identify it with an output node whose indegree=0.

a) For each added (output) node there is an internal node whose indegree = 1. All the routes through this node pass through its predecessor, therefore it is possible to identify it with its predecessor, by making all the edges outgoing from it to go from its predecessor. Thus the added node and edge are erased.

b) For each added (input) node there is in the resulting graph an internal node with outdegree=1. All the routes through this node pass through its successor, therefore it can be identified with it, eliminating the added node and edge.



input node with indegree > 0 identified with output node whose outdegree = 0

Figure 5.1

Networks representing computations are assumed to satisfy Assumption 1.

a) The input nodes have indegree 0.

b) The indegree of the computed nodes is 2.

c) The output nodes have arbitrary outdegree.

When networks satisfying assumption 1 are the building blocks in the recursive construction described in chapter 2, the resulting network satisfies assumption 1 as well. It is needed to prove only that each node constructed by identifying an output node in one subnetwork with an input node in the following subnetwork retains indegree 2. This is clear because each output node with indegree 2 is identified with an input node whose indegree is 0, to give a total indegree 2.

Note: The bounds on the number of computed nodes can be considered as bounds on networks satisfying assumption 1, the bounds on the number of edges are bounds on networks satisfying assumption 2.

Networks representing switching circuits are assumed to satisfy Assumption 2.

a) The input nodes have indegree 0.

b) The indegree of the computed nodes can be arbitrary.

c) The output nodes have outdegree 0.

In the recursive construction of networks satisfying assumption 2 it is possible to use SCs with input-indegree or output-outdegree ≥ 0. Only at the final stage of the construction must the output nodes have outdegree=0 and the input nodes must have indegree=0. Therefore in the recursive construction of networks satisfying assumption 2 it is possible to use networks satisfying assumption 3.

There can be at most a difference in size of N between networks satisfying assumption 2 and networks satisfying assumption 3. This is meaningful for small size networks which can serve as building blocks for larger SCs.

Assumption 3.

a) Each node (input node and computed node) may have an arbitrary indegree.

b) The outdegree of the output nodes ≥ 0.

c) The number of input nodes with indegree > 0 plus the number of output nodes with outdegree > 0 does not exceed N.

(c) ensures that it is possible to identify all the output nodes with outdegree>0 with input nodes whose indegree=0, and all the input nodes with indegree>0 with output nodes whose indegree=0. In what follows all the SCs drawn are assumed to have directed arcs from left to right and from bottom to top. i.e. the input nodes are at the bottom or at the left and the output nodes are at the top or at the right.

5.2 2-SCs, Recursive constructions of SCs and HCs.

By direct trial of all possible combinations it is possible to conclude:

1) The smallest size 2-SC satisfying assumption 1 has 2 computed nodes.
2) The smallest size 2-SC satisfying assumption 2 has 4 edges.

A complete crossbar (figure 5.2a) is a 2-SC satisfying assumptions 1 and 2. Starting from the non-minimal 2-SC of figure 5.2b with 5 edges and 3 computed nodes, and identifying output node O_1 which has indegree 1 with internal node E_1, we get the SC in figure 5.2c with 2 computed nodes and 4 edges. By identifying input node I_2 which has outdegree 1 with internal node E_1 (which is output node O_1) we get the 2-SC of figure 5.2d with 3 edges and 2 computed nodes. This SC is meaningful in the recursive construction only when output node O_1 (which has outdegree 1) in one stage is identified with input node I_1 (which has indegree 0) in the following stage and output node O_2 is identified with input node I_2.

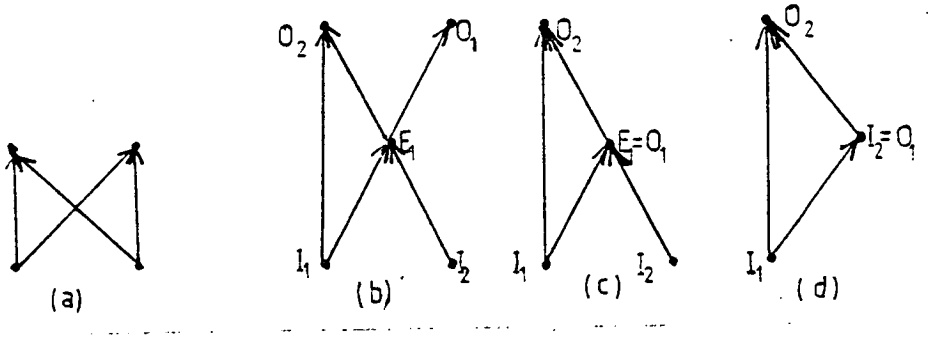


Figure 5.2 2-SCs

Theorem 5.1

The size of a SC satisfying assumption 3 satisfies

$$SC^{-}S(N) \leq 3\,N \log_2 N - (9/4)N + 2$$

$$= 4.754\,N \log_3 N - (9/4)N + 2$$

Proof

From corollary 2.2

for N a multiple of d:

$$SC^{-}S(N) \leq 2\cdot\frac{N}{d}\, SC^{-}S(d) - SC^{-}S(d) + d\cdot SC^{-}S(\frac{N}{d})$$

And for N=dq+r, r<d:

$$SC^{-}S(N) \leq 2\left\lfloor\frac{N}{d}\right\rfloor\cdot SC^{-}S(d)+SC^{-}S(r)+r\cdot SC^{-}S(\left\lceil\frac{N}{d}\right\rceil)+(d-r)SC^{-}S(\left\lfloor\frac{N}{d}\right\rfloor)$$

Use the SC of figure 5.2d as the building block for a SC which satisfies assumption 3.

d=2, $SC^{-}S(d)=3$, r=1 (or 0)

For N a multiple of 2:

$SC^{-}S(N) \leq 6(N/2)-3+2\cdot SC^{-}S(N/2)$

and for $N=2\cdot q+1$

$SC^{-}S(N)=6((N-1)/2)+SC^{-}S((N+1)/2)+SC^{-}S((N-1)/2)$

For N a power of 2, lemma 2.1 gives (with x=6, d=2, y=3)

$SC^{-}S(N)=3N\log_2 N-3N+3$

For N=2,3,4 the claim is clearly true (for N=3,4 use the SCs presented in the following section). Assume that for N'<N the claim is true.

If N is even then

$SC^{-}S(N)=3N-3+2\cdot SC^{-}S(N/2) \leq$ (using the induction hypothesis)

$3N-3+2\cdot(3N/2)\cdot\log_2(N/2)-2\cdot(9N/8)+4$

$$\leq 3N\log_2 N-9/4N+2$$

If N is odd then

$$SC^{-}S(N) \leq 2\cdot\frac{N-1}{2}\cdot 3+SC^{-}S(\frac{N+1}{2})+SC^{-}S(\frac{N-1}{2}) \leq$$

(using the induction hypothesis)

$$3N-3+3\frac{N-1}{2}\cdot\log_2(\frac{N-1}{2}) - \frac{9}{4}(\frac{N-1}{2}) +2 +3\frac{N+1}{2}\cdot\log_2(\frac{N+1}{2}) - \frac{9}{4}(\frac{N+1}{2}) +2 \leq$$

$$3N - 3 + \frac{3N}{2}(\log_2(N^2-1)) + \frac{3}{2}\log_2(\frac{N+1}{N-1}) - 3N - \frac{9}{4}N+4$$

For N>4, $\frac{3}{2}(\log_2(\frac{N+1}{N-1}))$ is less than 1, and the claim is proved.

Theorem 5.2

The size of a SC satisfying assumption 2 satisfies

$$SC^{-}S(N) \leq 3N \log_2 N - (5/4)N+2$$
$$=4.754\ N\log_3 N - (5/4)N+2$$

Proof

To get a SC satisfying assumption 2, at most N edges and nodes need to be added to the SC satisfying assumption 3.

This result, $3N\log_2 N-5/4N+2 = 4.754N\log_3 N-5/4N+2$ is better than the best result achieved using crossbars as building blocks for SCs:
$6N\log_3 N-(9/2)N+(9/2)$ (N a power of 3).

Theorem 5.3

The size of an UC (and of a HC) satisfying assumption 2 is bounded by:

$$UC^{-}S(N) \leq 1.5N\log_2 N+o(N\log N).$$

For N a power of 2

$$UC^{-}S(N) \leq 1.5N\cdot\log_2 N+N$$

Proof

By corollary 2.3, and using the SC of figure 5.2d as the building block, the size of UCs satisfying assumption 3 is bounded by:

$UC^{-}S(N) \leq 3\lceil N/2\rceil+2\cdot UC^{-}S(\lceil N/2\rceil)$

For N a power of 2

$UC^{-}S(N) \leq 3N/2 +2\cdot UC^{-}S(N/2)$

which by lemma 1.2 has a solution:

$$UC^{-}S(N)=1.5N\log_2 N,$$

and by adding at most N edges and nodes we get an UC (HC)

satisfying assumption 2 which satisfies:

$$UC^{-}S(N)=1.5N\log_2 N+N.$$

For N which is not a power of 2 the recurrence has solution $UC^{-}S(N) \leq 1.5N\log_2 N+o(N\log N)$.

5.3 Constructions

All the SCs were checked with the aid of a computer program.

Lemma 5.1

Let $SC^{-}N(N)$ be the minimal number of computed nodes in an N-SC with fanin 2.

$SC^{-}N(N+1) \geq SC^{-}N(N) + 3$.

Proof

By lemma 3.1 $SC^{-}N(N+1,N+1) \geq SC^{-}N(N,N+1)+2$
erasing an output node with outdegree 0 gives
$SC^{-}N(N+1,N+1) \geq SC^{-}N(N,N)+3$

5.3.1 3-Superconcentrators

By lemma 5.1 $SC^{-}N(3) \geq SC^{-}N(2)+3$. Minimal size 2-SCs have 2 computed nodes, therefore minimal size 3-SC satisfying assumption 1 has 5 computed nodes. There are five minimal 3-SCs satisfying both assumptions 1 and 2. They are depicted in figure 5.3, and there are 23 3-SCs satisfying only assumption 1. They are depicted in figure 5.4.

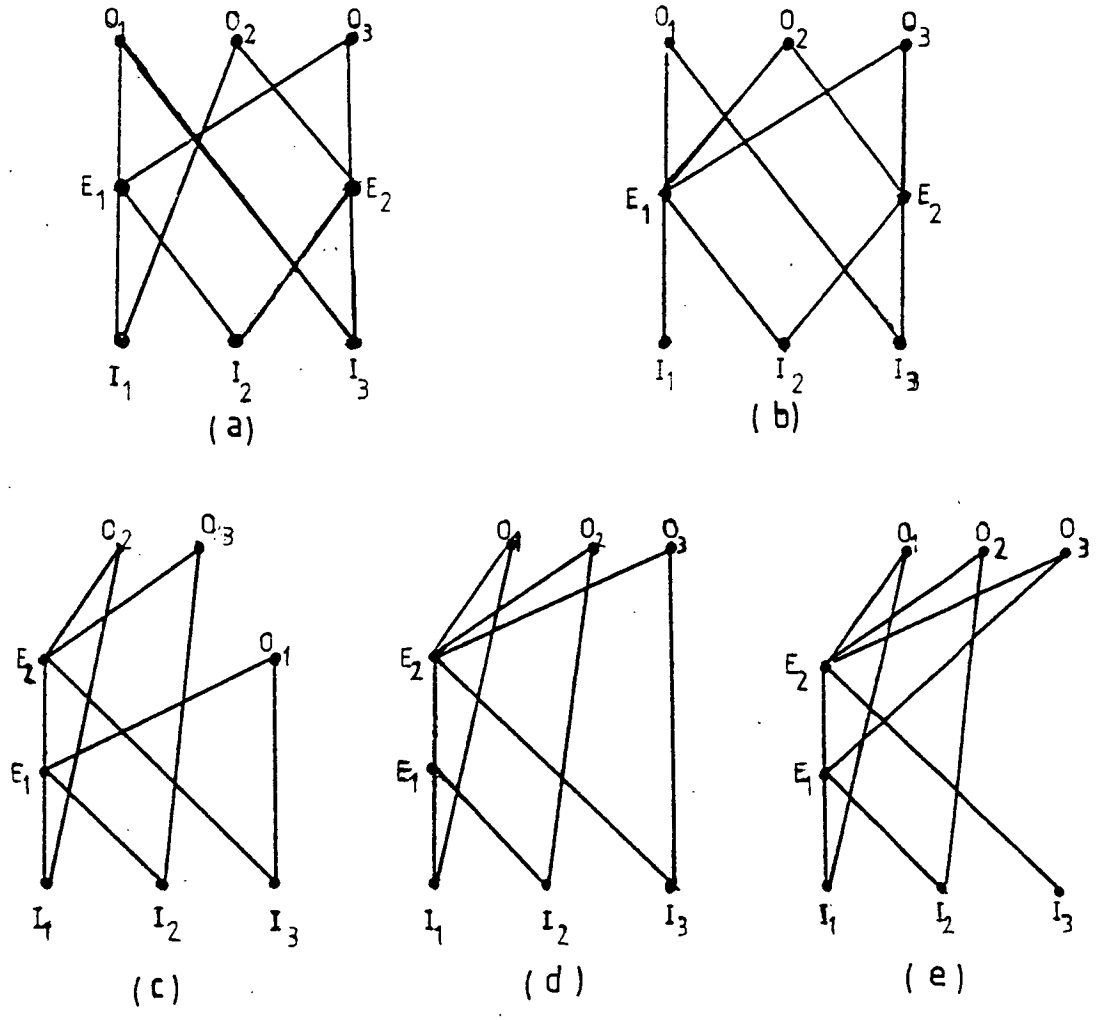
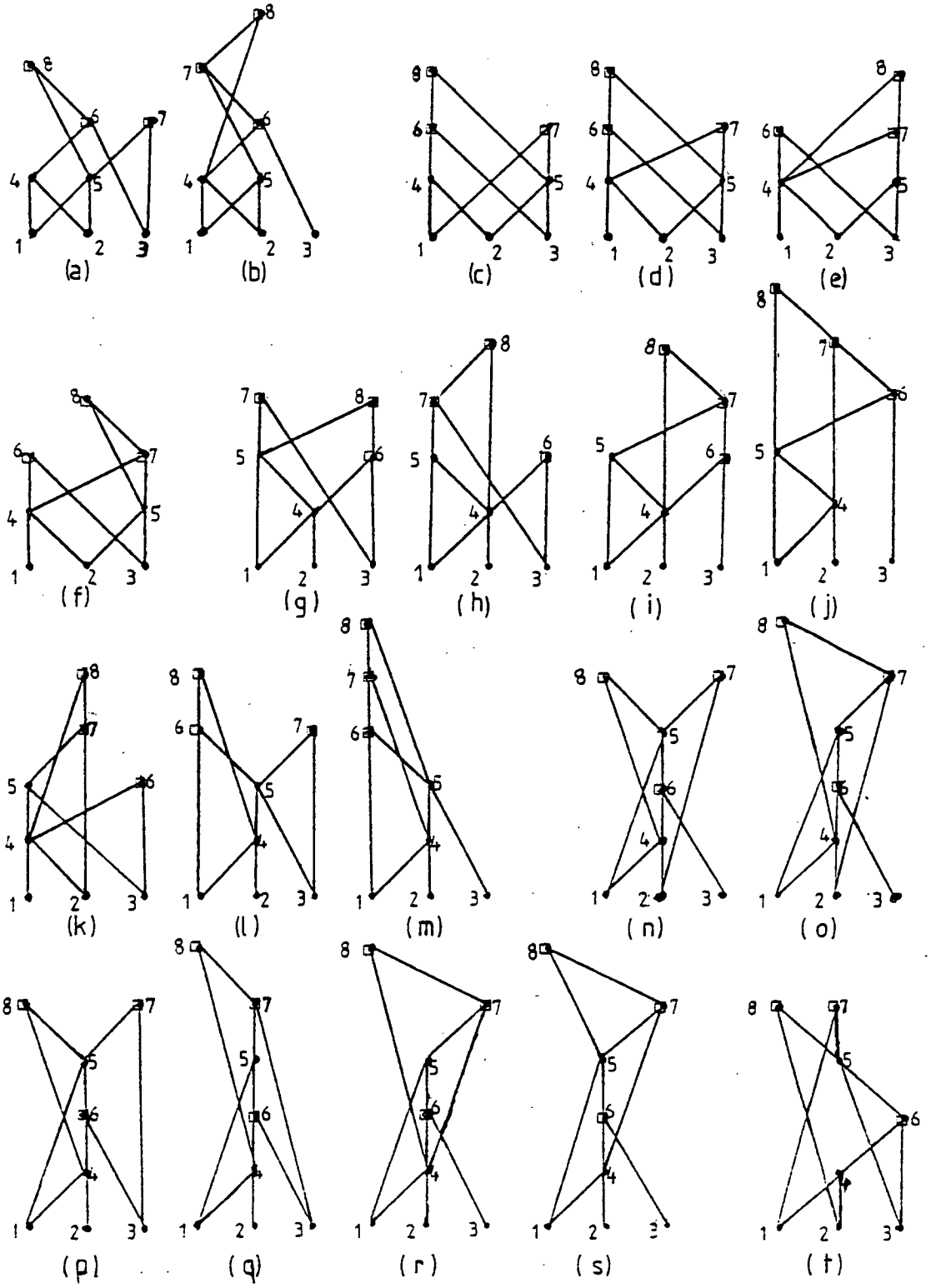


Figure 5.3 3-SCs satisfying assumptions 1 and 2

The following are SCs satisfying only assumption 1.

Nodes 1,2,3, are input nodes and nodes 6,7,8, are output nodes.



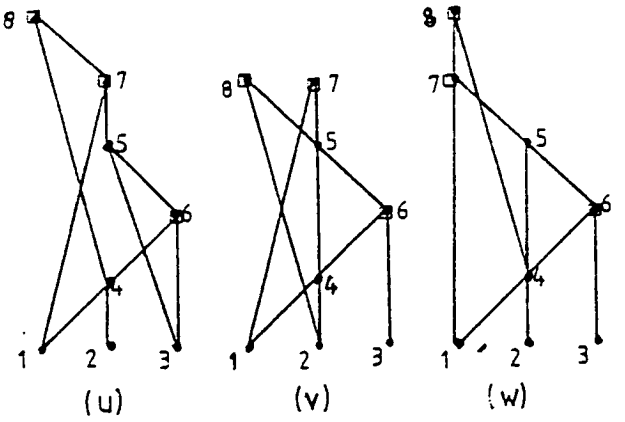


Figure 5.4 3-SCs satisfying only assumption 1

Lemma 5.2

There are only five 3-SCs satisfying assumptions 1 and 2, those that are depicted in figure 5.3.

Proof

Because of assumption 2, output nodes have outdegree 0 and input nodes have indegree 0, each output node has at least one internal node as predecessor in order to be connected to all three input nodes.

Claim: In a 3-SC satisfying assumptions 1 and 2 it is impossible to have 2 output nodes which have the same input node and internal node as predecessors.

This is because for the remaining 2 input nodes this internal node is a cut between the 2 input and 2 output nodes.

Corollary: The internal nodes have between them all three input nodes as predecessors;

For if the internal nodes succeed only 2 input nodes, all the output nodes will have to be connected directly to the third input node, two of the output nodes will have the same internal node as predecessor, contradicting the claim.

Consider the following possible cases:

a) The two internal nodes are at level 1.

By the corollary they have different predecessors, and by the claim at least one output node has the two internal nodes as predecessors. The two graphs satisfying this condition are 5.3a and 5.3b.

b) One internal node, say E_2, is at level 2, and therefore has E_1, the other internal node, and the third input node I_3, as predecessors. (Assume $Preds(E_1)=I_1,I_2$.)

If two input nodes, say a,b are connected to an internal node c, then we have to ensure node disjoint paths from a,b to any pair of output nodes. This is written as a,b -> not c. (i.e. for every pair of output nodes at least one of a,b is connected to one of them via a path not containing c.) Similarly for output nodes.

b.1) There is an output node at level 2.

It has E_1 and I_3 as predecessors, to ensure path from all the input nodes. The other two output nodes must each have I_1 or I_2 as predecessors to ensure I_1,I_2 -> not E_1 -> O_2,O_1, and I_1,I_2 -> not E_1 -> O_3,O_1. The only possible graph is 5.3c.

b.2) All three output nodes are at level 3.

Then all of them have E_2 as predecessor. To ensure node disjoint paths I_1,I_2 -> not E_1 (->O_1,O_2 or O_1,O_3 or O_2,O_3); two of the output nodes must have I_1 and I_2 as predecessors. (Say $Pred(O_1)=I_1, Pred(O_2)=I_2$) To ensure I_1,I_3 -> not E_2 (->O_2,O_3) and I_2,I_3 -> not E_2 (-> O_1,O_2) the third output node (O_3) can have either I_3 or E_1 as predecessors and the resulting graphs are 5.3d and 5.3e.

If we omit the indegree condition of assumption 1, then nine edges are sufficient (3x3 crossbar). This is also necessary under assumption 2':

Lemma 5.3

Let assumption 2' be assumption 2 where c is modified to:

c') The output nodes may have arbitrary outdegree.

Then the minimal 3-SC satisfying assumption 2' has 9 edges.

Proof

a) If there are no internal nodes then each output node is connected to each input node, or to an output node and two input nodes, yielding 9 edges (see for example figure 5.5).

b) If there are 2 internal nodes, we can assume without loss of generality that each computed node has indegree at least 2, otherwise it can be identified with its predecessor, reducing the number of edges by 1. But 2 internal nodes and 3 output nodes, each having indegree at least 2, contributes altogether 10 edges, contradicting the minimality.

c) Assume that there is one internal node. Such a SC can have 8 edges only if each of the output and the internal nodes has indegree 2, resulting in a graph satisfying assumption 1, which we have already proved to require 5 computed nodes.

Some 3-SCs satisfying assumption 2' with one internal node and nine edges are depicted in figure 5.6.

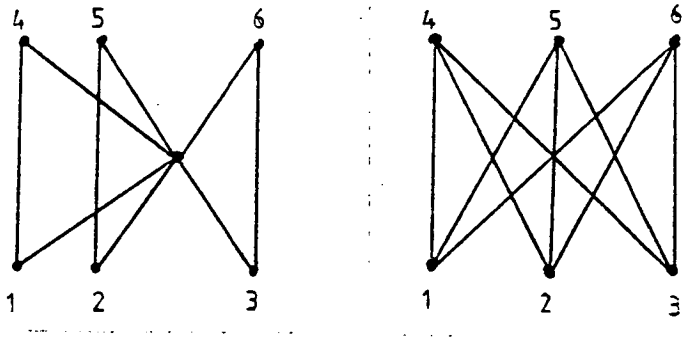


Figure 5.5 Minimal 3-SCs satisfying assumption 2

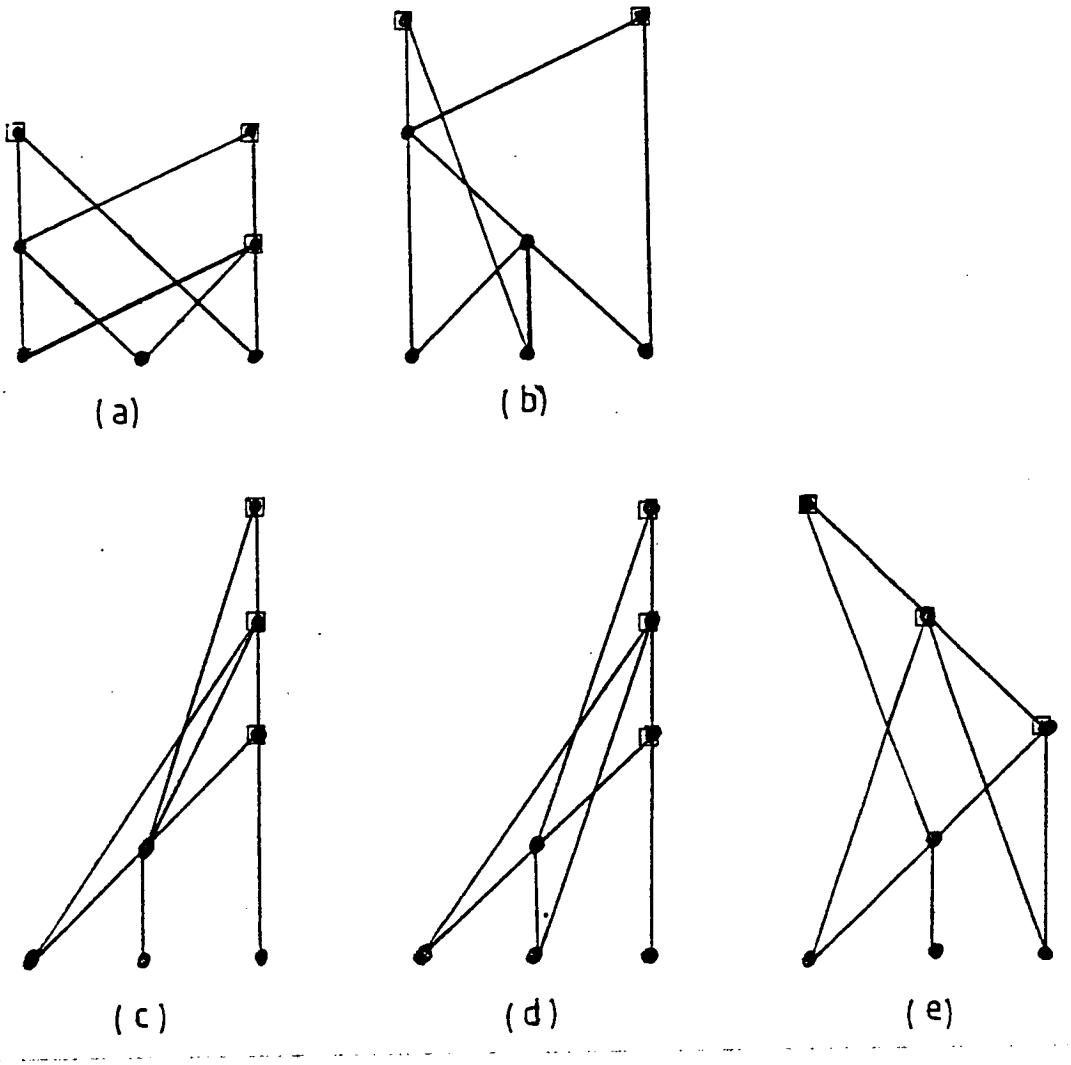


Figure 5.6 3-SCs satisfying assumption 2'

By identifying in figures 5.6c-5.6e an input node which has outdegree one with an internal node, we get 3-SCs satisfying assumption 3 with 8 edges. They can serve as building blocks in the recursive construction, yielding:

$SC^{-}S(N) \leq 2 \cdot 8/3\ N \log_3 N + o(N \log_3 N)$

$SC^{-}S(N) \leq 5.334\ N \log_3 N + o(N \log_3 N)$

which is better than the constant derived from 3x3 crossbars, but worse than the one derived from 2-SCs satisfying assumption 3. Note that by identifying the two input nodes which have outdegree 1 with the corresponding internal nodes in the SC of figure 5.6c, we get a 3-SC with 7 edges. It can serve as a subgraph in the construction of some larger SCs.

5.3.2 4-Superconcentrators

As a corollary from lemma 5.1 we get for 4 SCs satisfying assumption 1:

$SC^{-}N(4) \geq SC^{-}N(3) + 3$; $SC^{-}N(4) \geq 8$.

By manual search we found 7 different 4 SCs satisfying assumption 1 with 8 computed nodes. They are depicted in figures 5.7a-5.7g. A SC satisfying assumptions 1 and 2 with 9 computed nodes is depicted in figure 5.8.

In the following 1,2,3,4 are input nodes; 9,10,11,12 are output nodes.

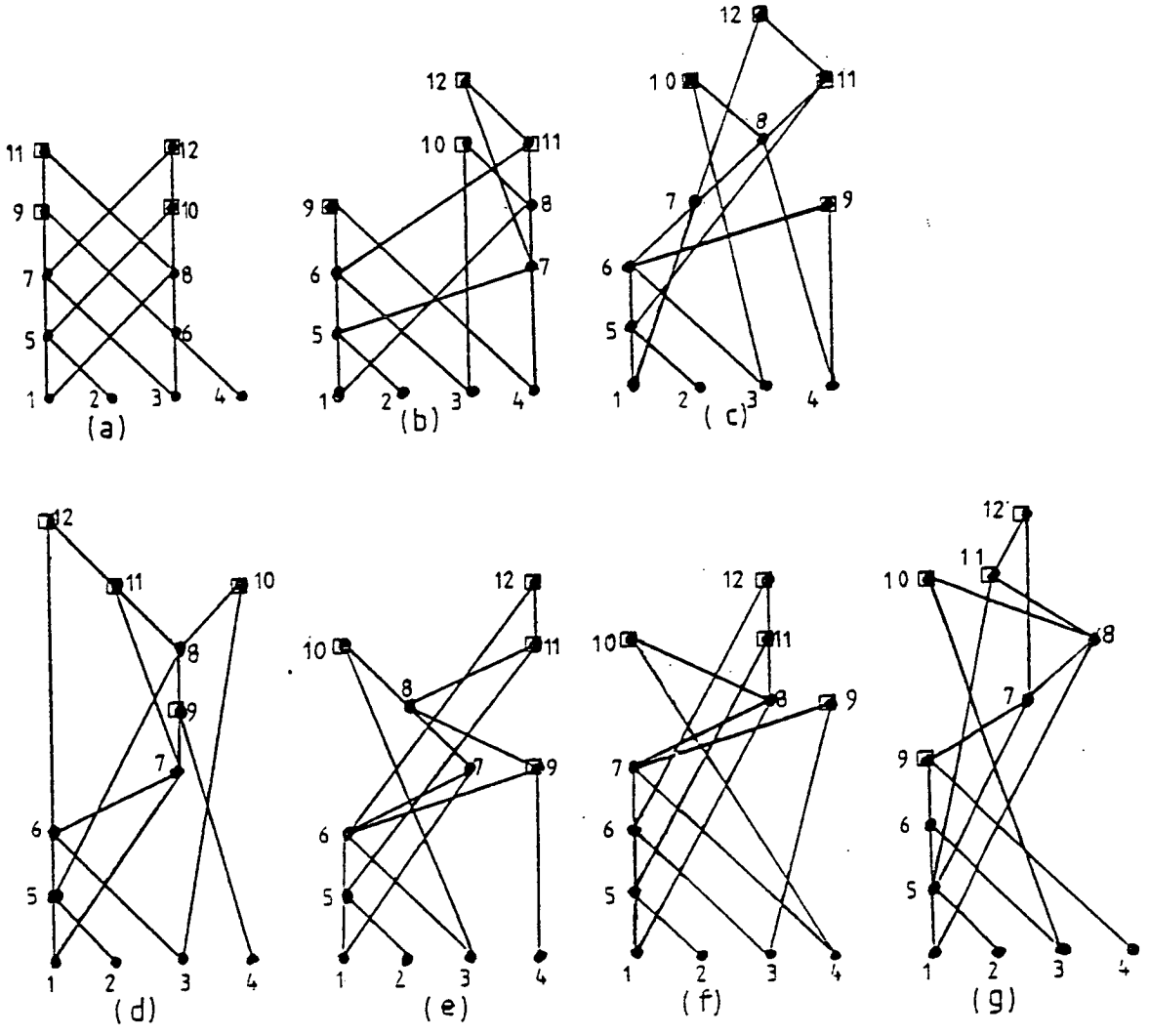


Figure 5.7 4-SCs satisfying assumption 1.

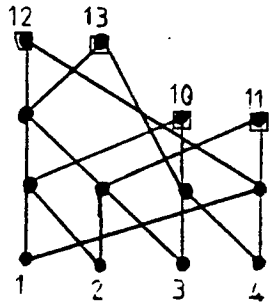


Figure 5.8
a 4-SC satisfying
assumptions 1 and 2

Using any of the SCs of figure 5.7 as building blocks for larger SCs when N is a power of 4 yields

$SC^{-}N(N)=8(2*(N/4)-1)+4\ SC^{-}N(N/4)$

$$SC^{-}N(N)=4\ N\ \log_4 N - 8/3\ N + 8/3$$

$SC^{-}N(N) = 2\ N\ \log_2 N - 2(N-1) - 2/3\ (N-1)$

compared with the minimal size SC satisfying assumption 1 which can be constructed from 2x2 crossbars and satisfies

$SC^{-}N(N) = 2(2*(N/2)-1) + 2\ SC^{-}N\ (N/2)$

$$SC^{-}N(N) = 2\ N\ \log_2 N - 2(N-1)$$

Using 4-SCs as building blocks for networks with N a power of 4 reduces the number of computed nodes only by a constant factor, but doubles the depth of the network from $2\ \log_2 N - 1$ to $4\ \log_2 N - 4$, if the SC of depth 4 (figure 5.7a) is used.

The minimal 4-SC satisfying assumption 2 known to me has 16 edges, i.e., the 4x4 crossbar. 4-SCs satisfying assumption 3 with 14 edges can be derived from the SCs in figures 5.7a, 5.7d, 5.7e, 5.7g. Using them as building blocks for larger SCs one gets:

$SC^{-}S\ (N) = 7\ N\ \log_4 N - 14/3\ N + 14/3 =$

$$3.5\ N\ \log_2 N - 14/3\ N + 14/3$$

which is better than the construction using 2x2 crossbars, but worse than the construction using 2-SCs satisfying assumption 3.

5.3.3 5-Superconcentrators

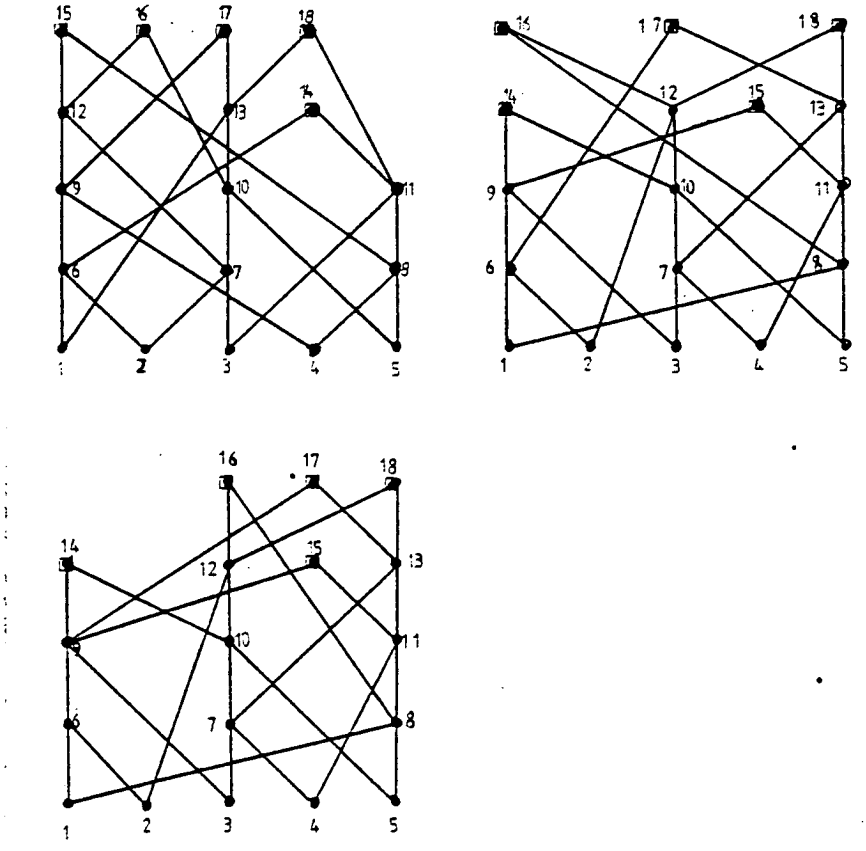


Figure 5.9 5-SCs satisfying assumption 1 with 13 computed nodes.

Figure 5.9 shows 3 graphs that are 5-SCs with 13 computed nodes under assumption 1. A lower bound of 11 follows easily from lemma 5.1:

$SC^{-}N(5) \geq SC^{-}N(4) + 3 = 11$

On the assumption that our list of 4-superconcentrators is exhaustive, we can improve this lower bound:

lemma 5.4

If the only 4-SCs satisfying assumption 1 are those depicted in figure 5.7, then there is no 5-SC satisfying assumption 1 with 11 computed nodes.

Proof

Assume that there is a 5-SC with 11 computed nodes. Then by lemma 3.1 there is a (4,5)-SC with 9 computed nodes. If there is such a (4,5)-SC with 9 computed nodes, erasing any output node with outdegree 0 should result in a 4-SC with 8 computed nodes, i.e, it should result in one of the SCs depicted in figure 5.7.

The proof proceeds by showing that a (4,5) graph with 9 computed nodes is not a (4,5)-SC. We check every possible combination of the number of output nodes with outdegree=0. By erasing one output node, or another we show that the resulting graph is not in the (possibly) exhaustive list.

1) Assume 5 output nodes have outdegree 0.
   There is no 4-SC with 8 computed nodes and with four outdegree-zero output nodes. If 1 is true, by erasing any output node in the (4,5)-SC we should get such a 4-SC. Contradiction.
2) Assume 4 output nodes have outdegree 0.
   Erasing any output node with outdegree 0 it is impossible to get a 4-SC with four outdegree-zero output nodes. Therefore the output node with outdegree > 0, say O_1, has either an internal node or two output nodes as direct successors. Erasing any output node with outdegree 0 should give a 4-SC with three outdegree-zero output nodes. O_1 cannot have an internal node as successor, because the only 4-SCs with such an output node have only two outdegree-zero output nodes. Therefore O_1 has at least 2 output nodes, say O_2, O_3 as direct successors. Erase the outdegree-zero output node different from O_2, O_3. The result should be a 4-SC with 8 computed nodes and one output node with two output nodes as direct sucessors. No such 4-SC exists.
3) Assume 3 output nodes have outdegree 0.
   3a) Assume none of the output nodes with outdegree zero has another output node as direct predecessor.

Erase any outdegree-zero output node to get a 4-SC with two outdegree-zero output nodes, none of them with an output node as predecessor. There is no such 4-SC.

3b) Assume one of the output nodes with outdegree zero, say O_3, has another output node, say O_1, as predecessr, and let O_2 be the second output node with outdegree > 0.

3b1) Assume O_1 has outdegree > 1 or O_1 has outdegree 1 and O_2 is not a predecessor of O_1.

Erase O_3. The result should be a 4 SC with 2 or 3 outdegree-zero output nodes, none of them with another output node as predecessor. No such 4-SC exists.

3b2) Assume O_1 has outdegree 1 and O_2 is its predecessor. Erase any outdegree-zero output node different from O_3. The result should be a 4-SC with 3 output nodes in a chain, i.e one immediately succeeding the other. No such 4-SC exists.

3c) Assume two of the outdegree-zero output nodes, say O_3, O_4, have an output node as direct predecessor and let O_5 be the third outdegree-zero output node.

Erase O_5. The result should be a 4-SC in which two outdegree-zero output nodes have an output node as predecessor. It can only be the SC of figure 5.7a. O_5 has only internal nodes as predecessors, therefore it is at level 3 at most, and the depth of the (4,5)-SC is 4. Consider again the (4,5)-SC, and erase either O_4 or O_3 (instead of O_5). The result should be a 4-SC of depth 4 with only one output node which has as predecessor another output node. No such SC exists.

3d) Assume 3 of the outdegree-zero output nodes, say O_3, O_4, O_5, have an output node as direct predecessor.

Then two of them, say O_4, O_5, have the same output node as predecessor. Erase O_3. The result should be a 4-SC with two outdegree-zero output nodes having the same output node as predecessor. There is no such 4-SC.

Therefore 3 is impossible.

4) Assume 2 output nodes, say O_4, O_5, have outdegree 0.

4a) Assume at least one of the outdegree-zero output nodes, say O_5, does not have another output node as predecessor. Erase O_5. The result should be a 4-SC with 3 output nodes with outdegree > 0. No such SC exists.

4b) Assume both O_4, O_5 have another output node as direct predecessor. Let $pred(O_5)=O_3$. Erase O_5. If O_3 has outdegree > 1 the result is a 4-SC with 3 output nodes with outdegree > 0, which is impossible. Therefore O_3 has outdegree=1 in the (4,5)-SC and the result is a 4-SC in which two output nodes, say O_1,O_2, have outdegree > 0 and only one of them has outgoing arc to another output node. The possible SCs are those depicted in figures 5.7d,e,g.

4b1) If $O_3=12$, $pred(O_3)=O_2=11$. Consider again the (4,5)-SC. Erase $O_4=10$ instead of O_5. The result should be a 4 SC in which there are 3 output nodes in a chain. No such 4 SC exists.

4b2) If $O_3=10$, $pred(O_3)$=an input node. Consider again the (4,5) SC and erase $O_4=12$ instead of $O_5=10$. The result should be a 4-SC in which there is an output node which has an input node as predecessor and an output node as sucessor. No such 4-SC exists.

5) Assume 1 output node has outdegree > 0.
Erase it. The result should be a 4-SC with at least 3 output nodes with outdegree > 0. Contradiction.

Therefore there is no 5-SC with 11 computed nodes.

Three 5-SCs satisfying assumption 1 with 13 computed nodes are depicted in figure 5.9. The best 5-SC satisfying assumption 2 known to me has 25 edges, i.e the 5x5 complete crossbar.

5.3.4 Larger size superconcentrators.

Assuming that the lower bound for 5-SCs satisfying assumption 1 is 12 nodes, the lower bound for 6-SC satisfying assumption 1 is 15 nodes. The best 6-SCs satisfying assumption 1 known to me have 18 computed nodes. Two 6-SCs with 18 computed nodes are drawn in figure 5.10. A SC satisfying assumption 2 with 33 edges, derived from 5.10 is drawn in 5.11.

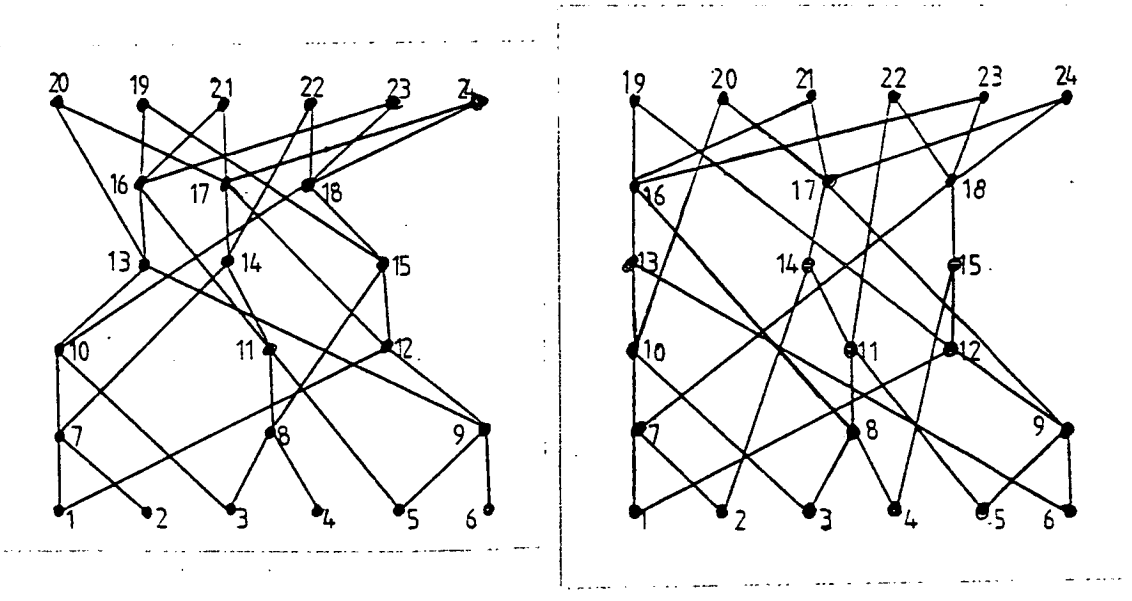


Figure 5.10 6-SCs satisfying assumption 1 with 18 computed nodes.

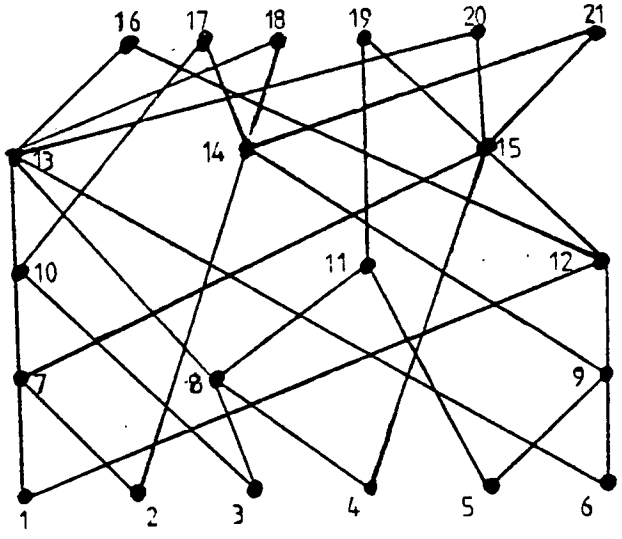


Figure 5.11
6-SC with 33 edges satisfying assumption 2'

Using the known small size SCs, it is possible to construct larger SCs, with the construction derived from theorem 2.2b:

7-SC The upper bound on the number of nodes in a 7-SC is 25, and upper bound on the number of edges in a 7-SC is 45, using the construction:

1, 2 SC, 2 SC, 2 SC
4 SC 3 SC
1, 2 SC, 2 SC, 2 SC

(where 1 stands for a 1 SC which is a single node).

8-SC $SC^{-}S(8) \leq 56$, $SC^{-}N(8) \leq 30$ using the construction:

2 SC, 2 SC, 2 SC, 2 SC
4 SC, 4 SC
1,1, 2 SC, 2 SC, 2 SC

In minimizing the number of edges, 4 SCs satisfying assumption 3 with 14 edges are used.

9-SC $SC^{-}N(9) \leq 37$ by using:

1, 2 SC, 2 SC, 2 SC, 2 SC
5 SC 4 SC
1, 2 SC, 2 SC, 2 SC, 2 SC

and $SC^{-}S(9) \leq 66$ edges using:

3 SC, 3 SC, 3 SC (each a 3x3 crossbar)
3 SC, 3 SC, 3 SC (each with 7 edges)
1,1,1, 3 SC, 3 SC (each a 3x3 crossbar)

10-SC $SC^{-}N(10) \leq 44$ using

2 SC, 2 SC, 2 SC, 2 SC, 2 SC
5 SC 5 SC
1,1, 2 SC, 2 SC, 2 SC, 2 SC

and $SC^{-}S(10) \leq 82$ edges using:

1, 3 SC, 3 SC, 3 SC
4 SC, 3 SC, 3 SC
1, 3 SC, 3 SC, 3 SC

11-SC $SC^{-}N(11) \leq 51$ nodes using:

1, 2 SC, 2 SC, 2 SC, 2 SC, 2 SC
6 SC 5 SC
1, 2 SC, 2 SC, 2 SC, 2 SC, 2 SC

and $SC^{-}S(11) \leq 93$ edges using:

2 SC, 3 SC, 3 SC, 3 SC
4 SC, 4 SC, 3 SC
1,1, 3 SC, 3 SC, 3 SC

12-SC $SC^{-}N(12) \leq 58$ nodes using:

2 SC, 2 SC, 2 SC, 2 SC, 2 SC, 2 SC
6 SC 6 SC
1,1, 2 SC, 2 SC, 2 SC, 2 SC, 2 SC

and $SC^{-}S(12) \leq 105$ using:

3 SC, 3 SC, 3 SC, 3 SC
4 SC, 4 SC, 4 SC
1,1,1, 3 SC, 3 SC, 3 SC

Summary of constructions

Size	SC⁻N	SC⁻S	SC⁻S (assumption 3)
2	2	4	3
3	5	9	8,7
4	8	16	14
5	13	25	
6	18	33	
7	25	45	
8	30	56	
9	37	66	
10	44	82	
11	51	93	
12	58	105	

5.3.6. Example of small hyperconcentrators

The same constructions can be applied to construct hyperconcentrators, for example, the 2-SC of figure 5.3d is an UC, and in figure 5.12a minimal 2-HC is drawn. Using the 2-UC of 5.2a or 5.2c and the 2-HC of 5.12, the 4-HCs of figure 5.13, satisfying assumption 1, with 5 nodes, are derived.

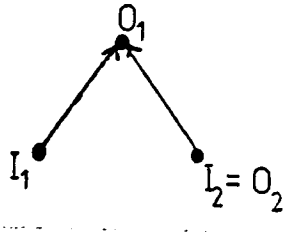


Figure 5.12
2-HC

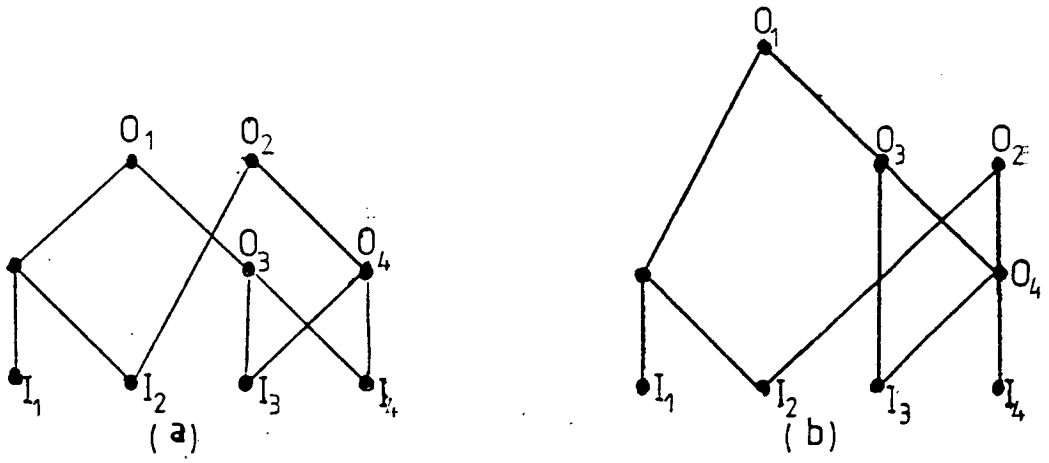


Figure 5.13 4-HCs

5.4 Using 3 SCs for computation

As was mentioned in the introduction, the graphs of linear programs for computing convolutions and the graphs for computing DFTs when N is a prime number are SCs, and those for computing DFTs are UCs (Valiant(1975a), Tompa(1978), relying on Dieudonne(1970)). The number of nodes in such graphs correspond to the number of additions performed by the computation. In this section we show that a successful search for an additions-efficient computation is sometimes possible, by first enumerating the graphs that have the required properties and then asking whether a suitable program can be based on them. The method appears feasible only for small computations, but these may, of course, be important subproblems of larger ones.

5.4.1 3-convolutions

$$\begin{pmatrix} A & B & C \\ B & C & A \\ C & A & B \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \end{pmatrix}$$

In the convolution represented by the above matrix, A,B,C are

regarded as constants, and the parameter to be minimized is the number of additions, excluding additions of constants. The obvious algorithm for this convolution requires 9 multiplications and 6 additions. Winograd's (1978) algorithm for convolution requires 4 multiplications and 11 additions.

Winograd's(1978) algorithm for 3 convolution:

$S_1 = y_0 + y_1$
$S_2 = y_0 - y_1$
$S_3 = y_1 - y_2$
$S_4 = y_2 - y_0$
$S_5 = y_2 + S_1$
$M_1 = ((A+B+C)/3)\cdot S_5$
$M_2 = ((2A-B-C)/3)\cdot S_2$
$M_3 = ((A+B-2C)/3)\cdot S_3$
$M_4 = ((A-2B+C)/3)\cdot S_4$

$S_6 = M_2 + M_3$
$S_7 = M_4 - M_3$
$S_8 = M_2 + M_4$
$F_0 = S_9 = M_1 + S_6$
$F_1 = S_{10} = M_1 + S_7$
$F_2 = S_{11} = M_1 - S_8$

Using the SC of figure 5.3c, and identifying

$I_1, I_2, I_3, O_1, O_2, O_3$ with

$y_0, y_1, y_2, F_2, F_1, F_0$ respectively

results in the following computation (see 5.14a)

$E_1 = 1\ y_0 + (A/C)\ y_1$
$E_2 = 1\ E_1 + (A/B)\ y_2$
$F_0 = (A-(CB/A))\ y_0 + (CB/A)\ E_2$
$F_1 = B\ E_2 + (C-(BA/C))\ y_1$
$F_2 = C\ E_1 + B\ y_2$

Which has only 5 additions (and 8 multiplications). Another computation graph for convolution using 5.3c is derived by identifying F_0, F_1, F_2 with O_1, O_2, O_3 respectively.

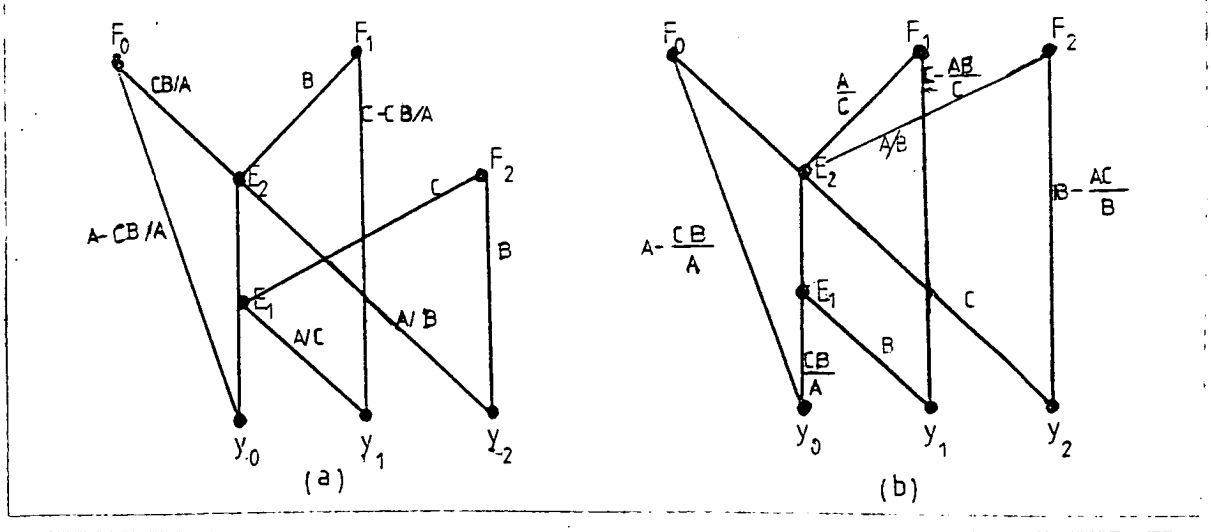


Figure 5.14 Convolutions using 3-SCs

The SC of 5.3d can compute the 3 convolution with the same number of additions and multiplications, for example, by the following computation: (figure 5.14b)

$E_1 = (CB/A)y_0 + By_1$

$E_2 = E_1 + Cy_2$

$F_0 = E_2 + (A-(CB/A))y_0$

$F_1 = (A/C)E_2 + (C-(AB/C))y_1$

$F_2 = (A/B)E_2 + (B-(AC/B))y_2$

Performing those operations on parallel processors, allowing fetch conflicts (see chapter 6) they can be performed using 4 processors in two multiplication steps and 3 addition steps. For example:

parallel step 1 multiplication: $p_1: T_1 \leftarrow (CB/A)y_0$

$p_2: T_2 \leftarrow By_1$

$p_3: T_3 \leftarrow Cy_2$

$p_4: T_4 \leftarrow (B-(AC/B))y_2$

parallel step 2 addition : $p_1: E_1 \leftarrow T_1+T_2$

parallel step 3 addition : $p_1: E_2 \leftarrow E_1+T_3$

parallel step 4 multiplication: $p_1: T_5 \leftarrow (A-(CB/A))y_0$

$p_2: T_6 \leftarrow (A/C)E_2$

$p_3: T_7 \leftarrow (C-(AB/C))y_1$

$p_4: T_8 \leftarrow (A/B)E_2$

parallel step 5 addition : p_1: $F_0 \leftarrow E_2+T_5$

p_2: $F_1 \leftarrow T_6+T_7$

p_3: $F_2 \leftarrow T_8+T_4$

5.4.2 3-DFT

In the 3-DFT represented by the matrix below w stands for the n`th (here the 3rd) root of unity. The powers of w are considered as constants and the parameter to be minimized is the number of additions, excluding additions of constants.

$$\begin{pmatrix} w^0 & w^0 & w^0 \\ w^0 & w^1 & w^2 \\ w^0 & w^2 & w^1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \end{pmatrix}$$

The obvious algoritm for this problem requires 4 multiplications and 6 additions. Winograd's(1978) algorithm for the 3 DFT requires 2 multiplications and 6 additions (and one additional multiplication by w^0) It is represented in figure 5.15a. Using the SC of figure 5.3c and identifying I_1, I_2, I_3, O_1, O_2, O_3 with y_2, y_1, y_0, F_0, F_2, F_1 respectively results in a computation for the 3 DFT which requires 3 multiplications and 5 additions, (and two additional multiplications by w^0) represented in 5.15b.

WINOGRAD`S ALGORITHM		SC`S ALGORITHM
$S_1=y_1+y_2$	$M_1=((w+w^2-2)/2)S_1$	$E_1=y_2+y_1$
$S_2=y_1-y_2$	$M_2=((w-w^2)/2)S_2$	$E_2=w^2E_1+y_0$
$F_0=S_3=S_1+y_0$		$F_0=S_3=y_0+E_1$
$S_4=S_3+M_1$		$F_1=S_4=E_2+(w-w^2)y_1$
$F_1=S_5=S_4+M_2$		$F_2=S_5=E_2+(w-w^2)y_2$
$F_2=S_6=S_4-M_2$		

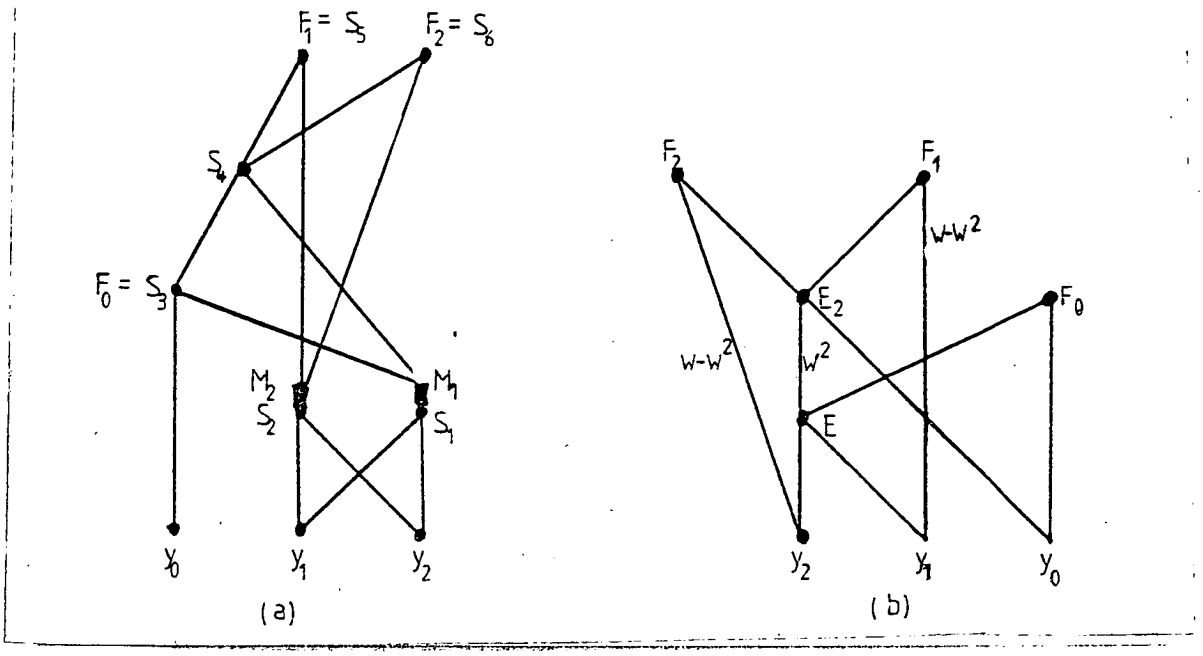


Figure 5.5 3-DFT using: (a) Winograd's algorithm, (b) 3-SC.

Note that Winograd`s algorithm can be done in one parallel multiplication step and 4 parallel addition steps using 2 processors, and the SC`s algorithm can be done in one parallel multiplication step and 3 parallel addition steps using 3 processors. Using the graph of 5.3d and identifying I_1, I_2, I_3, O_1, O_2, O_3 with y_0, y_1, y_2, F_2, F_1, F_0 respectively or identifying I_1, I_2, I_3, O_1, O_2, O_3 with y_2, y_1, y_0, F_2, F_1, F_0 respectively results in DFT computation which requires 5 additions and 5 multiplications.

5.4.3 5-DFT

The same 3-SC used above can improve the number of additions in computing the 5 DFT. (Here w is the 5th root of unity, $w=e^{(2\pi i/5)}=i\sin(2\pi/5)+\cos(2\pi/5)$. Let $u=2\pi/5$)

$$\begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \\ F_4 \end{pmatrix} = \begin{pmatrix} w^0 & w^0 & w^0 & w^0 & w^0 \\ w^0 & w^1 & w^2 & w^3 & w^4 \\ w^0 & w^2 & w^4 & w^1 & w^3 \\ w^0 & w^3 & w^1 & w^4 & w^2 \\ w^0 & w^4 & w^3 & w^2 & w^1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$$

The obvious algorithm to compute the 5-DFT requires 16 multiplications and 20 additions. Winograd's algorithm requires 5 multiplications and 17 additions. (The algorithm developed here requires 15 additions and 11 multiplications). As before, the powers of w are considered as constants, and the parameter to be minimized is the number of additions, excluding additions of constants. Consider the algorithm of Winograd, written on the left and represented by figure 5.16a.

WINOGRAD'S ALGORITHM		IMPROVED ALGORITHM
$S_1=y_1+y_4$		$S_1=y_1+y_4$
$S_2=y_1-y_4$	$M_3=cS_2$	$S_2=cy_1-cy_4$ $(=(y_1-y_4)c)$
$S_3=y_3+y_2$		$S_3=y_3+y_2$
$S_4=y_3-y_2$	$M_5=eS_4$	$S_4=ey_3-ey_2$ $(=(y_3-y_2)e)$
$S_5=S_1+S_3$	$M_1=aS_5$	$E_1=y_0+hS_1$
$S_6=S_1-S_3$	$M_2=bS_6$	$E_2=E_1+hS_3$
$S_7=S_2+S_4$	$M_4=dS_7$	
$F_0=S_8=S_5+y_0$		$F_0=S_8=\frac{1}{h}E_2+(1-(1/h))y_0$
$S_9=S_8+M_1$		
$S_{10}=S_9+M_2$		$S_{10}=E_2+(g-h)S_1$
$S_{11}=S_9-M_2$		$S_{11}=E_1+gS_3$
$S_{12}=M_3-M_4$		$S_{12}=fS_2-dS_4$
$S_{13}=M_4+M_5$		$S_{13}=dS_2+fS_4$
$F_1=S_{14}=S_{10}+S_{12}$		$F_1=S_{14}=S_{10}+S_{12}$
$F_4=S_{15}=S_{10}-S_{12}$		$F_4=S_{15}=S_{10}-S_{12}$
$F_2=S_{16}=S_{11}+S_{13}$		$F_2=S_{16}=S_{11}+S_{13}$
$F_3=S_{17}=S_{11}-S_{13}$		$F_3=S_{17}=S_{11}-S_{13}$

Where:

$a = (w+w^4)/4 + (w^2+w^3)/4 - 1$	$= (\cos u + \cos 2u)/2 - 1$
$b = (w+w^4)/4 - (w^2+w^3)/4$	$= (\cos u - \cos 2u)/2$
$c = (w-w^4)/2 + (w^2-w^3)/2$	$= i \sin u + i \sin 2u$
$d = (w^2-w^3)/2$	$= i \sin 2u$
$e = (w-w^4)/2 - (w^2-w^3)/2$	$= i \sin u - i \sin 2u$
$f = (w-w^4)/2$	$= i \sin u$
$g = (w+w^4)/2$	$= \cos u$

$h = (w^2+w^3)/2 \qquad = \cos 2u$

In wishing to minimize the number of additions, S_7 can be eliminated by replacing

$S_{12}=(c-d)S_2-dS_4 = fS_2-dS_4$

$S_{13}=dS_2+(d+e)S_4 = dS_2+fS_4$

In the resulting computation (see figure 5.16b) S_8, S_{10}, S_{11} are expressed in terms of y_0, S_3, S_1.

$S_8=y_0+S_1+S_3$

$S_{10}=y_0+(1+a+b)S_1+(1+a-b)S_3=y_0+gS_1+hS_3$

$S_{11}=y_0+(1+a-b)S_1+(1+a+b)S_3=y_0+hS_1+gS_3$

$$\begin{pmatrix}1 & 1 & 1\\ 1 & g & h\\ 1 & h & g\end{pmatrix}\begin{pmatrix}y_0\\ S_1\\ S_3\end{pmatrix} = \begin{pmatrix}F_0=S_8\\ S_{10}\\ S_{11}\end{pmatrix}$$

This computation has a similar form to the 3-DFT, and it is possible to match it to the graph of the 3 SC (see figure 5.16) to compute:

$E_1=y_0+hS_1$

$E_2=E_1+hS_3$

$S_8=(1/h)E_2+(1-1/h)y_0$

$S_{10}=E_2+(g-h)S_1$

$S_{11}=E_1+gS_3$

The resulting complete computation is presented to the right of Winograd's algorithm and is depicted in figure 5.16d. It has 15 additions and 11 multiplications.

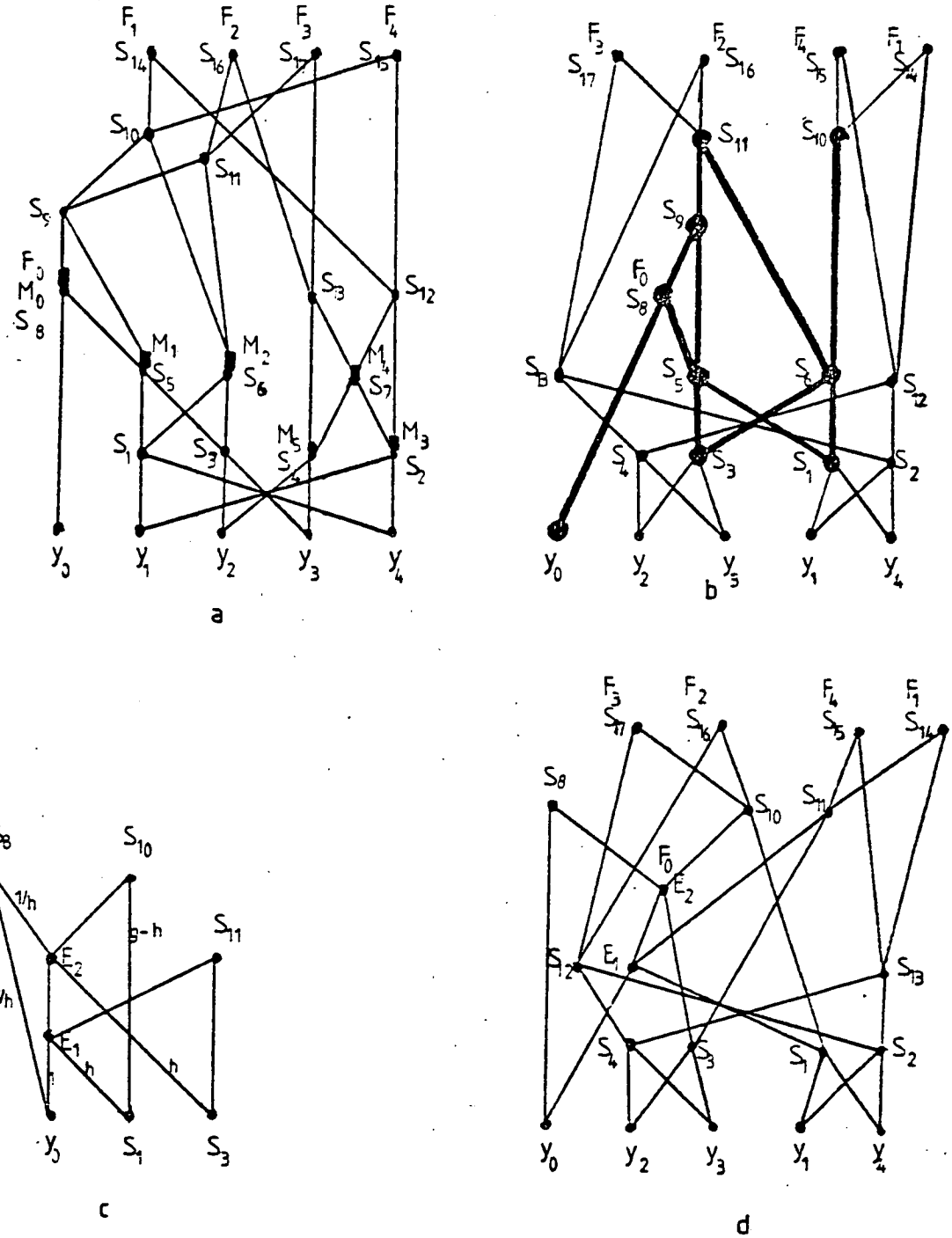


Figure 5.16 5-DFT

(a) Winograd's algorithm

(b) After eliminating S_7

(c) Using 3-SC to connect y_0, S_1, S_3 to S_8, S_{10}, S_{11}

(d) The graph of the improved algorithm

6. MODEL OF PARALLEL COMPUTATION

6.1 Introduction

In previous chapters, when graphs representing computations were considered, the main interest was in minimizing the number of computed nodes, i.e. in minimizing the number of binary operations performed. When the graph represented a computation on a sequential computer, it meant minimizing the number of sequential computation steps. When the graph represented a combinational circuit it meant minimizing the number of gates.

When considering parallel algorithms, we are interested in minimizing the number of parallel time units, (i.e. the depth of the graph), even at the expense of increasing the total number of computation steps or combinational gates. It is possible to envisage systems applications, (e.g. routing, sorting) where time is the most important factor, and one is ready to invest in "hardware", in order to reduce the time.

In the remaining chapters we shall be interested in the inherent parallel complexity of some computational problems, including routing in permutation networks. To this end, we present in this chapter the parallel model of computation assumed by the algorithms to be described. It has only a polynomial (in the input size) number of processors, each one with a small memory associated with it. The operation of the processors is synchronized, and each operation takes one unit of time. The word size is limited, so, though each processor may have a rich instruction set, the instructions operate only on words of restricted size.

The model assumes unrestricted parallel access to global memory. We show that this assumption does not introduce

surprisingly positive results, by proving that the parallel time required to compute any problem on a combinational circuit does not exceed the time required to compute the same problem on our model by more than a factor of O(logN).

6.2 Parallel RAC

The model of parallel computations used by the algorithms to follow is an extension of the RAC of Angluin and Valiant(1979), suggested by L. Valiant.

1) There are Π parallel processors, each one indexed uniquely. $\Pi = P(N)$, the number of processors is a polynomial function of N, the input size.

2) The processors share a common memory M.

3) Each processor has a local memory with a constant number of registers.

4) The operation of the processors is synchronized: each processor executes one instruction in one unit of time.

5) Each word in global memory and each register has size $\lambda = k \cdot \log_2 N$ for some constant $k \geq 1$. This allows the store to be of size polynomial in N, and restricts it to be of at most that size.

6) There is one program of size independent of N, stored in the local memory of each processor, and executed by all the processors simultaneously. The program can contain the symbol * , Which for each processor means its own numeric index.

7) All the processors have instruction set I.

The instruction set I

In describing the instruction set R(i,j) denotes the contents of the j'th register of processor i.

Any instruction that does not contain transfer of control (jump) can be prefixed by a conditional which will be described later.

References to the main memory

(a) FETCH (R(*,j), R(*,k))

(Assign to the local register R(*,j) the value of the main memory word whose address is stored in R(*,k).)

(b) STORE (R(*,j), R(*,k))

(Assign to the global-memory-word whose address is found in R(*,k) the value stored in the j'th register of the processor.)

assignments

(c) R(*,j) <- m (assignment of constant).

(d) R(*,j) <-R(*,k)

(assign to the local register j the contents of local register k.)

(e) R(*,j) <- *

(Assign to the j'th local register the index of the processor executing the instruction. We assume that a different value is stored in each local register as a result of this instruction. Since the length of each memory and register word is limited to $k \cdot \log_2 N$, this inherently bounds the number of processors to a polynomial in N.)

Arithmetic expressions

((f),(g),(h) are ... ; performed modulo 2^{λ})

(f) R(*,k) <- R(*,i)+R(*,j)

(g) R(*,k) <- R(*,i)-R(*,j)

(h) $R(*,k) \leftarrow R(*,i) \cdot R(*,j)$

(i) $R(*,k) \leftarrow R(*,i)/R(*,j)$ | (integer division)

Boolean expressions

(j) $R(*,i) \leftarrow \neg R(*,j)$

(Assign to the local register i the bitwise negation of local register j.)

(k) $R(*,i) \leftarrow R(*,j) \wedge R(*,k)$

(assign to the local register i the bitwise conjunction of registers j and k.)

Control statements

(l) HALT

(m) JUMP to m if $R(*,j)=0$

(transfer control to the instruction labelled m if $R(*,j)=0$.)

(n) if $R(*,j)=0$ then IDLE m else {any of instructions a-k)
if $R(*,j)=0$ then {any of a-k) else IDLE m.

(IDLE m means do nothing for m time units. The idle instruction enables one to synchronize the execution of various instructions in the program. In the algorithms presented this instruction is omitted and it is assumed implicitly when executing a conditional.)

8) Input and Output

The model assumes that at the beginning of execution there are N consecutive global memory cells which contain the input information, and at termination there are m specified memory locations which contain the output information. This enables us to discuss sublinear complexities of parallel algorithms.

9) The processors needed to execute the program are assumed to be active at the beginning of the program.

10) No simultaneous accesses to the same memory locations are allowed. If FETCH or STORE conflicts occur the algorithm

terminates, and the result is undetermined.

6.3 Some remarks on the model

The model is different from other models in that it restricts the number of processors to a polynomial in N, and it restricts the word length to be logarithmic in N. (Compare with Fortune and Wyllie(1978), Goldschlager (1978)).

The assumption that the processors are intiated in one time unit reduces by only an additive time factor of $\log_2 N$ the time to initiate all the processors (For example by an equivalent of the Fork instruction of Fortune and Wyllie (1978)). A processor which has to be idle until "required", can perform the following loop:

```
Loop:   FETCH (R(*,i),R(*,j))
        JUMP to Loop if R(*,i)=0
        JUMP to R(*,k)
```

(R(*,j) contains the address of a reserved memory location which contains 0 when the processor is not active.)

The model can be implemented using only 3 registers in each local memory, by dedicating in the global memory a special area for each processor. All the local memory instructions will have to fetch the arguments from the global memory and then to store the result.

If all the processors execute the same statments. i.e. all the conditional jumps perform the same instructions, then a program executed on this model can be implemented on the SIMD model of Flyn (1966), (Single Instruction stream, Multiple Data stream) by broadcasting all the instructions.

As each program has a fixed number of statements it can be simulated by a SIMD machine, increasing the time by a factor

proportional to the program size (i.e. a constant). For one original program step, each command in the program will be broadcast, but each processor will execute only the appropriate command conditioned on the contents of a local program counter.

There is nothing special about the instruction set chosen. Indeed, the set can be regarded as a parameter of the family of models. Also different conventions about STORE/FETCH conflicts may be worth considering.

6.4 A comment on Parallel Rac and Turing machines

In order to relate the Parallel RAC to other models of computation we first relate it to the model of combinational circuits.

Theorem 6.1

Any function computed by a Parallel-Rac(Π,k·logN,I) in T(N) time can be computed by a combinational circuit of size polynomial in N and depth O(T(N)·logN)

Proof

The operations performed by Parallel-RAC can be divided into those involving only local memory, and those involving STORE and FETCH to/from global memory.

We can assume without loss of generality that for each program statement, each processor performs the following steps:

a) FETCH from global memory.

b) COMPUTE in local memory (including memory transfers in local memory).

c) STORE to global memory.

For each such set of steps there corresponds one stage of the

combinational logic circuit. We consider the depth and the number of gates required to implement each such stage.

b) Each local memory operates as a RAM with a finite memory and restricted word size, therefore the operations can be performed using $O(\log N)^2$ gates and $O(\log \log N)^2$ depth. (Remember that the word length is $O(\log N)$). If we do not allow multiplication and division then each local memory operation requires $O(\log N)$ gates and $O(\log \log N)$ depth. If we add multiplication then $O(\log N \cdot \log \log N \cdot \log \log \log N)$ gates and $O(\log \log N)$ delay are sufficient. Adding division increases the delay to $O(\log \log N)^2$. (In both cases $O(\log N)^2$ gates and $O(\log N)$ depth are sufficient. (See Savage(1976)).

There are $\Pi(N)$ processors, therefore performing the local memory operation requires at most $O(\Pi(N) \cdot (\log N)^2)$ gates and $O(\log N)$ depth.

a), c) In both cases a processor can exchange information with a global memory location. In order to facilitate the description of the implementation we assume that in each local memory there are three special registers:

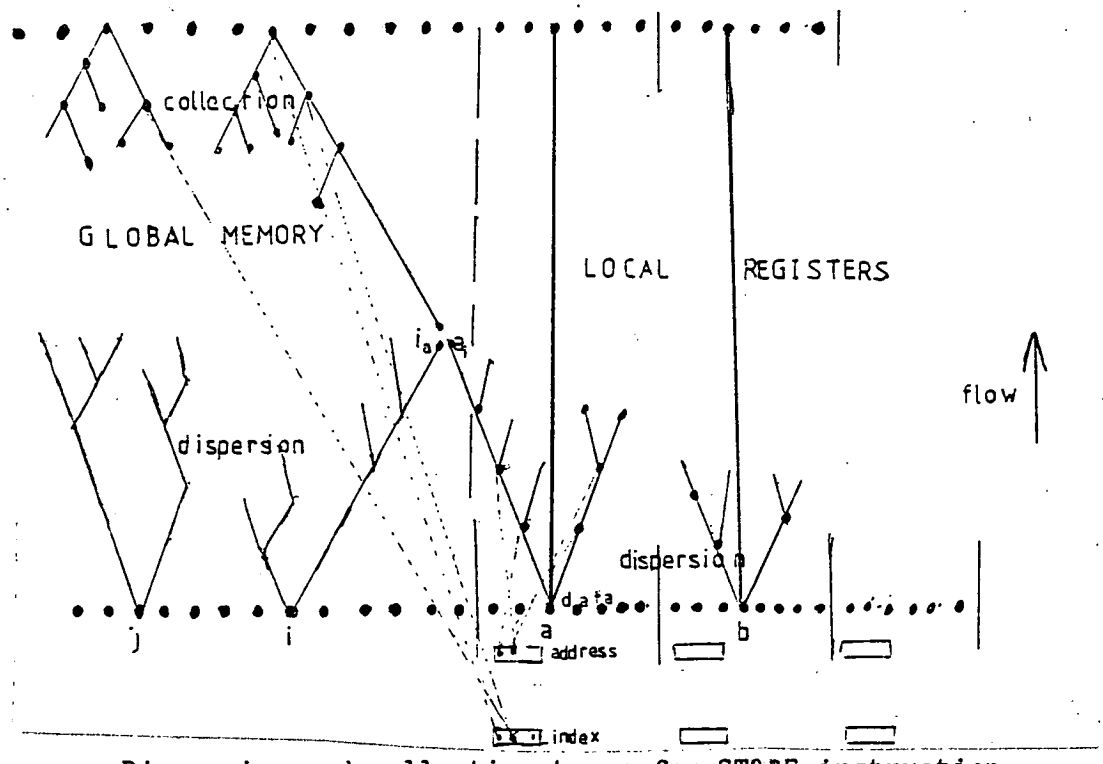
1) the "data-register", all the information to/from global memory paths through this register.
2) The (global) "address-register", it contains the address of global memory to be accessed by the processor.
3) The "index-register", it contains the index of the processor.
((R(*,j) <- * is now R(*,j) <- index-register(*).)

The circuit of a STORE instruction.

We describe loosely the combinational logic circuit implementing a store instruction. We sometimes use terminology borrowed from switching circuits; a switch can be represented by a logic gate in which one input corresponds to the data line and one input corresponds to the control line.

Assume that for each global memory location there are "dispersion" circuits in the form of binary trees. Each such tree has π leaves, one for each processor. (We denote the a'th leaf in the tree of global memory location j by j_a). For each bit in the data-register there is a dispersion tree, in which each leaf corresponds to a global memory position. The j'th leaf of processor a is a_j. The j_a and a_j leaves are connected by an appropriate gate. Further assume that for each data-register the control gate prevents the information from passing from the root of the tree to the leaves until some gate is "activated". Each bit in the address register "enables" half the branches corresponding to its position. In this way the information to be stored will be transferred only to the leaf which corresponds to the global memory location where the information should be stored. The information from the roots corresponding to the global memory location is transfered to all the leaves. For each global memory location there is a "collection tree" in which the roots correspond to memory location after the store is completed. Each bit in the index-register(s), "and"ed with the result of the address register, disables the information from leaves of the collection tree (of the global memory) which do not correspond to its own processor. (It disables the leaves only in the tree to which the data was transferred). Thus if some data was stored in a global memory location, at the end of this substage the gate corresponding to this memory location will contain the new data. Each global memory location in which no store occurred will have the old information stored in it. Each local memory location contains the same data as before.

For FETCH instructions there will be a dispersion tree for each global memory location and a collection tree for each local memory location. (Note that in this implementation fetch conflicts are allowed for free).



Dispersion and collection trees for STORE instruction

Figure 6.1

The number of gates required by the address register bit in position i is $exp_2(i)$ multiplied by O(logN), the size of the word to be stored. Summing over i, it requires at most $O(exp_2(k \cdot log_2 N) \cdot logN)$ gates. The depth is O(logN). The number of combinational logic gates required to implement the index-register switching function corresponds to the number of bits of the index register, times the data-register size times the memory size. (For each global memory collection tree only one path to the root has to be "blocked"). Therefore the number of combinational logic circuits is $O(exp_2(k \cdot log_2 N) \cdot log^2 N)$ and for all the processors and all the global memory locations it is $O(exp_2(k \cdot log_2 N) \cdot \Pi(N) \cdot log^2 N)$ gates. The depth of the circuit is clearly O(logN). Therefore each stage of the logic circuit corresponding to one unit of time on our Parallel-RAC reqires a number of gates polynomial in the input size, and O(logN) depth.

Therefore any function that can be computed in O(T(N)) time on Parallel-RAC can be computed with $O(T(N) \cdot exp_2(k \cdot log_2 N) \Pi(N) log^2 N)$ gates and $O(T(N) \cdot logN)$ depth.

Corollary

Any function computed in T(N) time by Parallel-RAC is computed by a TM in work space $O(T(N)\cdot\log N)$.

Outline proof

Borodin (1977) proved that any set computed uniformly by a combinational circuit in depth d(N) can be computed by a Turing-Machine in space d(N). We showed that any program computed by Parallel-Rac in T(N) time is computed by a uniform combinational circuit of depth $O(T(N)\cdot\log N)$, therefore it can be computed by a TM in space $O(T(N)\cdot\log N)$.

7 PARALLEL OPERATIONS ON ORIENTED LISTS: A PARALLEL PARITY LABELLING ALGORITHM

In the following chapters a parallel algorithm for routing in permutation or superconcentration networks which are constructed recursively as described in chapter 2 is developed. This chapter presents the basic parallel algorithm which appears as a subroutine in the other algorithms, the parallel parity labelling algorithm for lists. It improves on a randomized algorithm for the problem suggested by L. Valiant. The algorithm presented here performs transitive closure operations on an "oriented" list which may be closed (circular). Algorithms for related problems on lists are presented as corollaries of this algorithm. At the end of the chapter, oriented graphs, data structures preserving orientation and orientation routines are discussed. .

7.1 Definitions

A parity labelling of a sequence is assigning a binary value (say 0,1) to the elements of the sequence, so that every element in the sequence gets a value different from the value of its neighbours. The two values will be referred to as opposite to each other.

Let a sequence of items be represented by a doubly-linked-list in direct access memory. For each item there is a data block, in which there are, among others, two fields PT_0 and PT_1 which are pointers to the item's neighbours in the sequence. The data block of an item at the end of an open sequence contains a special symbol ϕ in one of its PT_i fields.

Sometimes we wish to discuss the connection between two neighbouring items as a separate concept. We therefore define the link between two such items to consist of the pair of pointers

pointing to each other.

It is possible to view each item as composed of two half-items, indexed 0 and 1, each one containing a pointer to one of the item's neighbours and corresponding to one of the item's links. Each link is composed of two parts, corresponding to its two half-items.

In an undirected sequence the data block of an item does not contain information about which of its neighbours precedes it and which one follows it. In a directed sequence this information exists.

In an unoriented sequence each PT_i field of an item "points" to the whole data block of the corresponding neighbour, and there is no information as to which of the PT_i fields in the neighbour's data block "points back" to the item. If such information is available, the sequence is oriented. In an unoriented sequence both of the item's neighbours may be considered as pointing at its 0th half.

In an oriented sequence, the orientation field of a pointer contains a 0, (the orientation equals 0) if the neighbour pointed at by this pointer is pointing back to the item from its 0th half, and it contains a 1 if the back pointer is from the 1st half.

In a directed sequence the direction field contains a 0 if the pointer points to the item's predecessor, and it contains a 1 if the pointer points to the item's successor.

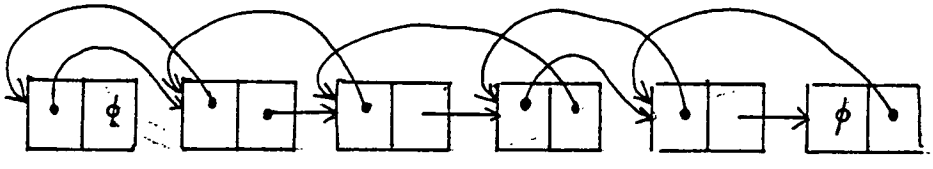
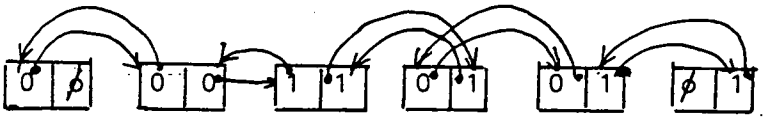
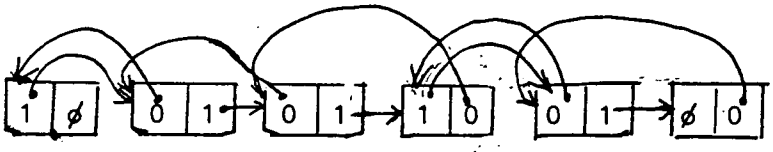


Figure 7.1 an undirected and unoriented list



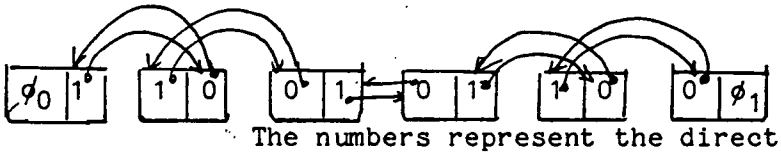
The numbers represent the orientation

Figure 7.2 An undirected and oriented list



The numbers represent the direction

Figure 7.3 a directed and unoriented list



The numbers represent the direction

(The orientation fields are not shown)

Figure 7.4 A directed and oriented list

The parity labelling algorithm can perform one of the following tasks:

1) Parity label the items of the sequence;
   Assign the same labels to the two halves of each item and opposite labels to the two halves of each link.

2) Parity label the links of the sequence;
   Assign opposite labels to the two halves of each item and the same labels to the two halves of each link.

3) Assign direction to the sequence
   Assign opposite labels to the two halves of each item and opposite labels to the two halves of each link.

7.2 Parity labelling an oriented sequence

In what follows a parallel parity labelling algorithm for an oriented list is described. The algorithm uses N processors, O(N) space, and runs in O(logN) time. In this algorithm read conflicts are not allowed, i.e. no two processors can read from the same memory location at the same time.

The orientation of any two pointers pointing at the same item is opposite. This fact enables the algorithm to avoid two accesses to the same memory location at the same time. In many implementations the two half-items are stored in different memory locations, and so it is possible to access the two half-items simultaneously. Otherwise the two accesses would have to be made at different times.

A unique priority is associated with each item. This priority is used to prevent deadlock, which would occur otherwise if two items with the same priority tried to propagate different parity labelling. This priority may be the item's number or it may be some other feature of a particular implementation of the algorithm.

The algorithm assumes that one processor is assigned to each item. (When half-items can be accessed separately it is possible to assign a processor to each half item). Arbitrary parities, and unique priority are given to each item. At the i'th iteration of the algorithm's main loop, each item compares its priority with the priorities of items at distance 2^i from itself, and acquires the highest priority among the two compared and its own priority, and acquires parities compatible with those of the item with the highest priority among those three items. At this iteration all the items whose distance from the item with the highest priority does not exceed 2^i, receive the highest priority and compatible parities. Hence in no more than $\log_2 N$ iterations, all the items get consistent parities.

Each item requires the following fields in global memory:

Input information

PT_0 PT_1 pointers to the item's neighbours.

OT_0 OT_1 Orientation for PT_0,PT_1 respectively.

AR_0 AR_1 Priority (arbitration) fields.

Result fields

IP_0 IP_1 Item parity labels.

LP_0 LP_1 Link parity labels.

DP_0 DP_1 Direction parity labels.

Working storage

PW_0 PW_1 Pointer fields

OW_0 OW_1 Orientation associated with PW_0,PW_1.

The algorithm computes the three parities independently of each other. If only one of the parity labels is required, the other two may be discarded. On reading the algorithm for the first time it may be helpful, therefore, to ignore LP and DP, say.

Depending on the particular implementation of the algorithm, each pair of fields for all the items may be stored in a two dimensional matrix, for example PT[0:N-1 , 0:1], where PT(i,j) contains the value of PT_j belonging to item i. In this case each pointer is the index in the matrix of the neighbour-item. There may be more fields associated with each item, they are not shown here.

It is assumed that each processor has local memory which contains the output fields, the working storage fields and the priority fields of the item to which it is assigned. Global memory locations are represented by upper case characters and local memory locations by lower case characters. Each reference to a field-name means field-name(*), the field in the data block of the item assigned to the processor. Opposite value is denoted

by a line above the corresponding variable. A field followed by subscript with a line above the subscript denotes the field in the other half of the item's data block. A line above a field means the opposite value to the value in that field. e.g $x_{\overline{T}}=x_0$; if $x_0=1$ then $\overline{x_0}=0$; if $x_0(y_1)=0$ then $\overline{x_0(y_1)}=1$.

Algorithm 7.1
Parallel parity labelling

input: $[PT_t, OT_t, AR_t, IP_t, LP_t, DP_t \ t=0,1]$
{The input parameters are the field names in the data block of each list item. The input consists of doubly linked list(s) of items, with the pointers to the item's neighbours stored in PT_0, PT_1 and their orientation in OT_0, OT_1 respectively. The priority of each item is stored in AR_0, AR_1. Some value 0 or 1 is stored in IP_0, LP_0, DP_0.}

Assign one processor to each item.

!each processor par-do
In each data block define (or allocate) PW_t, OW_t t=0,1.

(Initiation)
$PW_0 \leftarrow PT_0$; $PW_1 \leftarrow PT_1$
$OW_0 \leftarrow OT_0$; $OW_1 \leftarrow OT_1$
$IP_1 \leftarrow IP_0$; $LP_1 \leftarrow \overline{LP_0}$; $DP_1 \leftarrow \overline{DP_0}$

Copy the relevant fields to the processor's local memory
and denote their local occurrences as small letters.
(end initiation)

For i=0 step 1 until $\lceil \log_2 N \rceil - 1$ do 1
(Main-loop)
for j=0,1 do 2
for k=0,1 do 3 (*)

```
        if pw_j≠φ and ow_j=k then do 4                          (*)
            (Access the data block                              (*)
            indicated by pw_j and ow_j.)                        (*)
            (update-function)
            if AR_k(pw_j)>ar_j then do 5
                ar_j <- AR_k(pw_j)
                ip_j <- (if i=0 then ¬IP_k(pw_j) else IP_k(pw_j))
                lp_j <- (if i=0 then LP_k(pw_j) else ¬lp_k(pw_j))
                dp_j <- ¬DP_k(pw_j)
            end do 5
            (end-update-function)

            (update-pointers)
            ow_j <- OW_k(pw_j)
            pw_j <- PW_k(pw_j)
            (end-update-pointers)

        end do 4                                                (**)
    end do 3                                                    (**)
end do 2

(update-consistency)
if ar_0>ar_1 then j=0
else if ar_1>ar_0 then j=1.
if ar_0≠ar_1 then do 6
    ip_¬j <- ip_j
    lp_¬j <- ¬lp_j
    dp_¬j <- ¬dp_j
    ar_¬j <- ar_j
end do 6 else
    if ip_¬j≠ip_j then error stop
        (Cyclic list with odd number of elements)
(end-update-consistency)

copy the processor's local memory to global memory
(end-main-loop)
```

end do 1

end par-do.

end of procedure.

Let i_j denote the j'th half item of item i.

The distance along the sequence is defined for half items as follows:

$dist(i_j,(PT(i_j))_{OT(i_j)})=1$

If $dist(i_j,k_r)=n_1$ and $dist(\overline{k_r},m_n)=n_2$

then $dist(i_j,m_n)=n_1+n_2$.

If $dist(i_j,k_r)=n$ then $dist(\overline{i_j},k_r)=-n$.

and the non directed distance between two items i,k:

$dist(i,k)=\min(|dist(i_j,k_r\ ;\ j,r=0,1)|)$

and the distance for links:

$dist(i_j,(PT(i_j))_{OT(i_j)})=0$

$dist(i_j,\overline{i_j})=-1$

If $dist(i_j,k_r)=n_1$ and $dist(\overline{k_r},m_n)=n_2$

then $dist(i_j,m_n)=n_1+n_2+1$.

The algorithm's correctness is shown with the help of the following two lemmas.

Lemma 7.1

At the begining of the i'th iteration of the main loop each pair of PW_j, OW_j either point to the end (PW_j contains ϕ) or they point to an item at distance 2^i and to a link at distance 2^i-1 from the item and link corresponding to their own half item.

Proof

The proof is by induction on i.

For i=0 the claim is true by the input conditions.

Assume the claim is true for i iterations, then at the i'th

iteration, at update-pointers , the pair PW_j OW_j is set to point to an item at distance 2^i+2^i and to a link at distance 2^i-1+2^i-1+1, or PW_j is set to ϕ.

Lemma 7.2

Let q be the item with the highest initial priority in the sequence. At the begining of the i'th iteration of the main loop, all the items whose distance from q does not exceed 2^i-1 has the highest priority and parities compatible with the initial parities of q.

Proof

The proof is by induction on i. It is true before the 0'th execution of the loop.
Assume the claim true for i. At the i'th iteration of the loop, each item whose distance from q is j, $0 \leq j \leq 2^i-1$ and which already has the highest priority and parities compatible with those of q by the induction hypothesis, accesses, at update-function, the item at distance 2^i from itself to which it points by lemma 7.1, and updates the parities and priority of that item, so that before the i+1 execution of the algorithm each item whose distance from q does not exceed $2^i+2^i-1=2^{i+1}-1$ has the highest priorities and compatible parities.

Theorem 7.1

The parity labelling algorithm assigns compatible parities, in parallel, to any list which is not circular with an odd number of elements. It requires O(N) space and O(logN') time, where N is the number of elements in all the lists, and N' is the number of elements in the longest list.

Proof

Each iteration of the main-loop requires O(1) time. By lemma 7.2 after the $\log_2 N'$ iteration of the loop each item whose distance from the highest priority item in the list, q, does not exceed $2^{i+1}-1$ has parities compatible with those of q, i.e each item in the lists has compatible parities. The space requirement is obvious.

7.3 Generalization to other operations on lists.

The basic scheme of algorithm 7.1 can be applied to various other problems. In each such algorithm, at the update-function step, the data block of the j'th half of each item n is updated as a function of itself and of the information in the $OW_j(n)$'th half of the $PW_j(n)$'th item. At the update-pointer step the pointers (and accumulated information) are updated to point to items at distance 2^{i+1}. At the update-consistency step the two halves of each item update each other and become consistent with one another.

The exact function of the priority field differs according to the particulars of the problem at hand. A rough classification is as follows.

a) Sometimes when the solution to the problem is single valued and wrap around causes no problems, we may not need the priority field at all. For example, suppose a value VL(*) is associated with each item in the list and we wish to find the maximal value.

In all the following algorithms the parity fields IP,LP,DP are not required.

Algorithm 7.2

Finding the maximum.

Execute algorithem 7.1 with the following modifications:

input $[PT_t, OT_t, VL_t, RS_t]$

(The input includes a value VL stored in the data block of each item. The priority fields AR_i are redundant.
Additional field RS in each half item is added to contain the result.)

Replace Initiation step, line 3 by

RS_0 <- VL

RS_1 <- VL

Replace update-function by

rs_j <- max(rs_j, $RS_k(pw_j)$)

Replace update-consistency with

RS_0, RS_1 <- max(RS_0, RS_1)

The algorithm gives the correct result also when the list is circular with odd number of elements.

b) When the problem is not single valued we need priorities to do arbitration. If wrap-around causes no particular difficulties we do not need to keep track of the item's position relative to some specific (high priority) item. The parity labelling algorithms are examples of such algorithms. When a value VL is associated with each item, finding the index of an item with the highest value is another such problem.

c) When the list is assured to be an open list, it is possible to eliminate the priority; the implicit assumption in this case is that the end items have the highest priority, and the functions are computed relative to the direction of computation.

Algorithm 7.3

Finding the (weighted) position of an item in an open list

Execute algorithm 7.1 with the following modifications:

input $[PT_t, OT_t, (WT), RS_t]$

(The field RS in each half item contains the computed results.)

Replace initiation line 3 by

$RS_0 \leftarrow 1$; $RS_1 \leftarrow 1$ ($RS_0 \leftarrow WT$; $RS_1 \leftarrow WT$)

Replace update-function by

if $pw_j \neq \phi$ then $rs_j \leftarrow rs_j + RS_k(pw_j)$

Erase update consistency.

At the end of the algorithms RS_0, RS_1 contain the index of the item (or the sum of weights of items) (starting from 1) when the list is traversed from one or another end, and the length of the list is RS_0+RS_1-1. If a direction is found as well, then the distance of the item from the begining and the end of the list may be found. Starting with given weights WT associated with each item, and RS_0, RS_1 containing weights, RS_0, RS_1 will give the weight of the items from the list's ends to the item (inclusive), and $RS_0+RS_1-WT_0$ will give the weight of the list's items.

d) When the list can be circular, and wrap-around would cause difficulties the functions may be computed in each direction up to the item with the highest priority in that direction. (when the list is open the highest priority items are the end ones.) This restriction ensures that each function is computed only once across each link. The function of each pair of items is computed once, except the function of each item with the highest priority item which is computed twice in circular lists, and can be eliminated at the end.

algorithm 7.4

Finding the weighted distance to the highest priority item.

Execute algorthm 7.1 with the following modifications:

input: $[PT_t, OT_t, WT_t, RS_t]$

{Assume that the weighted distance of each item to its neighbour is stored in both halves of each link, in WT, before initiation. Additional fields: RS to store the result and TM, a working field are assumed in each item.

Before the i'th iteration of the main loop TM contains the weighted distance to the item at distance 2^i, and RS contains the weighted distance to the highest priority item up to distance 2^i-1 (in the direction of its half item).}

Add: Define TM_t, t=0,1 in each data block.

Replace initiation line 3 by

$RS_0 \leftarrow 0$

$RS_1 \leftarrow 0$

$TM_0 \leftarrow WT_0$

$TM_1 \leftarrow WT_1$

Replace update-function by

if $PW_{\overline{k}}(pw_j) \neq \emptyset$ and $AR_{\overline{k}}(pw_j) > ar_j$ then do 5

$ar_j \leftarrow AR_{\overline{k}}(pw_j)$

$rs_j \leftarrow TM_j + RS_{\overline{k}}(pw_j)$

end do 5

Add to update-pointer

$tm_j \leftarrow tm_j + TM_{\overline{k}}(pw_j)$

At the end of the algorithm each half-item contains the distance to the item with the highest priority in its direction. The nondirected distance to the item with the highest priority is ND:

ND <- (if $AR_0 > AR_1$ then S_0
else if $AR_1 > AR_0$ then S_1
else if $S_0 < S_1$ then S_0
else S_1.

The weight of all the links in the sequence is S_0+S_1.

7.4 Oriented and non oriented lists and graphs

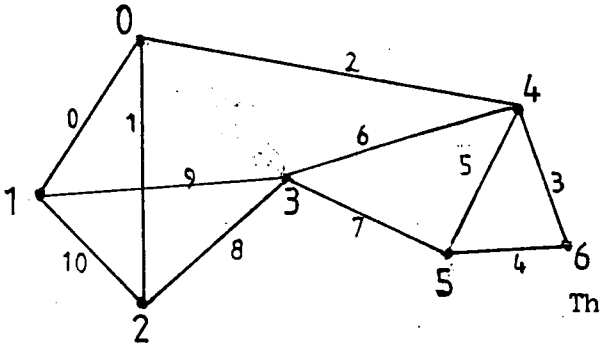
All the lists dealt with in the sequel are oriented lists, however a suitable data structure is needed, which preserves the orientation information.

In the routing algorithm, there is a graph G with N nodes and maximal degree d. Assume that the graph is represented by node-edge incidence lists (Neil). For each node there is a list of the edges incident with it and for each edge there is a list of the two nodes incident with it. A data block is associated with each list element. Each block in the list of the nodes points to an edge incident with the node, and each block in the edge list points to a node at its end. If the data structure is oriented, each data block in the nodes' lists also contains the index (0 or 1) in the list of the edge pointing back at it, and each edge contains with the node name the index in the nodes' list of the block pointing back at it.

An alternative representation nodes adjacency lists (Nal) consists of having a list for each node. Each element of such a list points to the list of one of the nodes adjacent to it. In this structure each edge is represented by a link, by the data blocks corresponding to its two ends and pointing at each other. If the list is oriented then each such data block contains, in addition, the position in the list pointing back at it.

When the degree of the graph is d it is possible to replace the lists representation by a matrix representation. The nodes

represented by a matrix V[0:N-1, 0:d-1], where each row corresponds to the list of one node. The edges are represented by a matrix E[0:E-1, 0:1]. The various representations are shown in figure 7.5. Each matrix position is a data block containing fields PT,OT,etc.. the fields in the matrix are abbreviated, for example the PT field in the (i,j) block of the matrix V is abbreviated to V⁻PT(i,j) or to PT(i,j) if the matrix is understood from the context. Being equivalent from the algorithmic point of view, any of those data structures will be used. They will be refered to as <u>oriented graph</u> representation.



The graph G

<u>unoriented NAL matrix</u>

node \ position				
0	4	2	1	ø
1	0	3	2	ø
2	1	0	3	ø
3	1	2	5	4
4	0	3	5	6
5	6	4	3	ø
6	5	4	ø	ø

<u>Unoriented NIEL matrices</u>

node \ position				
0	0	1	2	ø
1	0	9	10	ø
2	10	1	8	ø
3	9	8	7	6
4	3	5	6	2
5	7	5	4	ø
6	4	3	ø	ø

edge \ position		
0	0	1
1	0	2
2	4	0
3	6	4
4	5	6
5	4	5
6	4	3
7	5	3
8	3	2
9	1	3
10	1	2

<u>unoriented representation</u>

Oriented Nal matrix

node \ position				
0	4_0	2_1	1_0	ϕ
1	0_2	3_0	2_0	ϕ
2	1_2	0_1	3_1	ϕ
3	1_1	2_2	5_2	4_2
4	0_0	3_3	5_1	6_1
5	6_0	4_2	3_2	ϕ
6	5_0	4_3	ϕ	ϕ

Oriented Niel matrices

node \ position				
0	0_0	1_0	2_1	ϕ
1	0_1	9_0	10_0	ϕ
2	10_1	1_1	8_1	ϕ
3	9_1	8_0	7_1	6_1
4	3_1	5_0	6_0	2_0
5	7_0	5_1	4_0	ϕ
6	4_1	3_0	ϕ	ϕ

edge \ position		
0	0_0	1_0
1	0_1	2_1
2	4_3	0_2
3	6_1	4_0
4	5_2	6_0
5	4_1	5_1
6	4_2	3_3
7	5_0	3_2
8	3_1	2_2
9	1_1	3_0
10	1_2	2_0

Oriented representations

In each block we wrote PT_{OT}

Figure 7.5 Various graph representations

Algorithm 7.1 and the derived algorithms assume that the lists are oriented. If the lists are not oriented, but fetch conflicts are allowed, the above algorithms are applicable with the statements marked (*) replaced by:

access the data block indicated by pw_j
if $pw_j=pw_{\bar{j}}$
    then k <- j
    else
        if $PW_0(pw_j)=(*)$ then k <- 0 else k <- 1

and the statements marked (**) erased.

It is possible to orient the sequence with a similar procedure at the initiation step.

With the above oriented graph representation, lists may be considered as graphs with maximal degree two.

When fetch conflicts are not allowed, and the graph does not contain parallel edges, orientation may be found in O(1) time using N·d processors. The procedure can be adapted to operate using (N/y)d processors in O(y) time. The required space is $O(N^2)$.

Assume that a graph is represented by an unoriented Nal matrix. Let Q be an NxN matrix. The orientation procedure will store in Q(i,j) the orientation of the pointer pointing at node i from node j.

Orientation procedure

Each processor indexed (*) is represented here by $i=a_d(*)$ and $j=b_d(*)$. (a,b are defined in chapter 2).

```
!each i,j par do 1 (!each proc par-do 1)
   Q(PT(i,j),i) <- j
   OT(i,j) <- Q(i,PT(i,j)).
end par do 1
end of procedure.
```

Lists may be oriented easily when parallel edges exist, say, by having a global policy that if the two half-items are pointing to the same item, then the orientation of each half equals its own index (0 or 1), and then applying the above orientation procedure to the remaining links which represent a graph without parallel edges.

If the graph contains parallel edges, orientation may be found in $O(\log_2 d)$ time using $N^2 \cdot d$ processors and $O(N^2 d)$ space. The orientation procedure uses a matrix Q(NxNxd) to store the orientation. Initially $Q=\phi$ (and the weight=0). If the k'th index of the j'th item points to item i, it is stored in Q(i,j,k), and a weight 1 is stored in the weight matrix WT'. For each fixed value of i and j there is a list with d items. After finding the index of each active pointer in each such list, the pointers are condensed to the first locations of each (third dimension) column, so that if there are r edges between node i and node j, the position of the r' th pointer from j to i is stored in Q(i,j,r'), pointing to the r' th pointer from i to j whose position is stored in Q(j,i,r').

Each processor, indexed (*) is represented by (i,j,k), $k=b_d(*)$,

$j=b_N(a_d(*))$, $i=a_N(a_d(*))$

Orientation procedure (outline)

```
!each i,j,k par-do 1
   Q(i,j,k) <- ∅
   WT'(i,j,k) <- 0
end par do 1.

!each proc i,k (j=0) par do 2
   t <-PT(i,k)
   Q(t,i,k) <- k
   WT'(t,i,k) <- 1
end par do 2
```

For fixed value of i and j the positions of k define a sparse array. We make this array into a list:

```
!each i,j,k par do 3
   PT'0(i,j,k) <- "i,j,k-1"
   if k=0 then PT'0(i,j,k) <- ∅
   OT'0(i,j,k) <- 1
end par do 3
```

Call weighted count (algorithm 7.3)
input [PT',OT',WT',RS']
The index of non-empty entries is stored in RS'(i,j,k)

```
!each i,k par do 4
   t <- PT(i,k)
   r <- RS'(t,i,k)
   Q(i,j,r) <- k

   OT(i,j) <- Q(j,i,r)
End par do 4
end of procedure.
```

8 PARALLEL ALGORITHMS FOR EDGE COLOURING BIPARTITE GRAPHS

8.1 Introduction

Fast parallel algorithms for edge colouring bipartite graphs are presented in this chapter. The first algorithm assumes that the maximal degree of the graph, d, is a power of 2, and gives an edge colouring in $O(\log E \cdot \log d)$ time. The second algorithm is a parallel implementation of the Gabow and Kariv(1978) edge colouring algorithm, using "typed recolour". It requires $O(d \cdot \log^2 N)$ time for any d. For $d=2^k \cdot C$ this algorithm requires only $O_k(C \cdot \log^2 N)$ time. Recently a new sequential edge colouring was devised by Gabow and Kariv (to appear). It was brought to our attention by N. Pippenger who noted its amenability to parallel implementation. With the aid of routines used in the other algorithms we give an implementation using $O(\log^2 N \cdot \log d)$ time for any d.

The last two parallel algorithms use the first one as a subroutine, and for d a power of 2 they reduce to the first one. The basic operation performed in all those algorithms is splitting a graph into two graphs, so that the degrees of the resulting graphs are small enough. In the first algorithm the degrees satisfy the condition that after $\lceil \log_2 d \rceil$ iterations each of the resulting graphs has degree 1, and may be assigned one colour. There are $2^{\lceil \log_2 d \rceil}$ graphs, so the graph is coloured in that number of colours.

8.2 Graph partition

Let G be a bipartite graph with V nodes, E edges and maximal degree d. We want to partition the edges of G into two subgraphs, each with the same node set as G, and each with a maximal degree $\lceil d/2 \rceil$.

Assume that the edges of G are partitioned into paths, so that each node has at most t occurrences on paths. (Each time a node v' appears on the path

$$v_0 \overset{e_0}{\text{----}} v_1 \overset{e_1}{\text{----}} \cdots \quad v_{n-1} \overset{e_{n-1}}{\text{------}} v_n$$

it is considered as one occurrence of the node on the path. In closed cycles $v_0=v_n$ is counted as one occurrence of that node.)

The graph partition relies on the following lemma:

Lemma 8.1

Let the edges of bipartite graph G be partitioned into paths, so that each node v has at most t(v) occurrences on paths. If the edges of the paths are parity labelled, each edge labelled 0 is assigned to subgraph 0 and each edge labelled 1 is assigned to subgraph 1, then the degree of each node v in each of the resulting graphs is at most t(v) and the maximal degree of each of the resulting graphs is $\max_v(t(v))$.

Proof

The graph is bipartite, therefore each closed cycle has an even number of edges, and it is possible to parity label the paths. Each occurrence of a node on a path contributes, at most, one to its degree in each of the resulting subgraphs.

When a graph is partitioned, it is possible to create new data structures corresponding to its subgraphs. Alternatively, it is

possible to store the subgraphs in place, using a scheme which allows one to distinguish the subgraphs.

Let $D=\exp_2\lceil \log_2 d \rceil$, and let the graphs be represented by oriented Neil matrices V[0:N-1 , 0:D-1] and E[0:E-1 ,0:1], where each matrix position is a data block.

Algorithm 8.1
Graph partition

input [V⁻(PT,OT ,D); E⁻(PT,OT); D]
(Fields ending with PT,OT are pointer and orientation fields respectively. d is stored in V⁻D. E⁻(L⁻IP,L⁻PT,L⁻OT,L⁻AR) are variables used for calling the parity labelling routine.)

Assign one processor to each edge.

```
!each proc par do 1
   if V⁻D≤1 then exit
   (It is impossible to split the graph any more)

   (creating paths)
   (make edges whose pointers in a node's list have orientation 2k
   and 2k+1 adjacent on a path.)
   for i=0,1 do 2
       j <- E⁻PT(*,i)
       k <- E⁻OT(*,i)
       t <= (if (k mod 2)=0 then 1 else -1)
       if k+t>V⁻D then E⁻L⁻PT(*,i) <- ϕ else do 3
           E⁻L⁻PT(*,i) <- V⁻PT(j,k+t)
           E⁻L⁻OT(*,i) <- V⁻OT(j,k+t)
       end do 3
       E⁻L⁻AR(*,i) <- *
   end do 2
   (end creating paths)
   (The path parameters are stored in E⁻L⁻ )
```

Call Parity label for items (Algorithm 7.1)

input [E^-L^-PT,E^-O^-PT,E^-L^-AR,E^-L^-IP]

(Each path is considered as a list of edges, and its items, the edges are parity labelled. The parity is stored in E^-L^-IP.)

(store subgraphs)

(The index of each position in each node's list is divided by 2; if the corresponding edge was labelled 0, the data block is stored in the first half of the list corresponding to the original subgraph, and if it was labelled 1, it is stored in the second half.)

Copy to local memory the data block pointing to the edge from the nodes' matrix.

```
for i=0,1 do 4
    j <- E⁻PT(*,i)
    k <- E⁻OT(*,i)
    k_a <- a_D(k)
    k_b <- a_2(b_D(k))
    if E⁻L⁻IP=0 then do 5
        V(j,p_D(k_a,k_b)) <- v(j,k) (from local memory)
        E⁻OT(*,i) <- p_D(k_a,k_b)
    end do 5 else do 6
        V(j,p_D(k_a,k_b)+D/2) <- v(j,k) (from local memory)
        E⁻OT(*,i) <- p_D(k_a,k_b)+D/2
    end do 6
    V⁻D <- ⌈(V⁻D)/2⌉
end do 4
(end store subgraphs)
```

(The edge pointed at from position (j,k) in a node's list belongs to the $a_D(k)$ subgraph. It is the $b_D(k)$ edge in the list of that subgraph. After the partition it becomes the $\lfloor (b_D(k))/2 \rfloor = a_2(b_D(k))$ edge in the list of the $2 \cdot a_D(k)+E^-L^-IP$ subgraph.)

end par-do 1

end of procedure

Various schemes for choosing neighbouring edges on paths may be implemented by changing the create-paths step. Similarly, the storage scheme may be different. It is possible to use a Nal matrix, in which case each node is divided into $2\lceil d(v)/2 \rceil$ nodes, and the parity labelling is performed on the links which represent the edges. If the original structure of the graph has to be preserved, it is possible to store the original orientation of each edge-end in the nodes' lists. Another possibility is to keep the original orientation, and store the index of the subgraph to which the edge belongs. An approach similar to the last one is presented in Lev-Pippenger-Valiant(to appear).

Theorem 8.1

Algorithm 8.1 splits graphs whose maximal degree is d into two subgraphs in O(log E) time. The resulting subgraphs have maximal degree $\lceil d/2 \rceil$.

Proof

The algorithm calls "Parity-labelling" once. This requires O(log E) time, as the maximal list length may be E. The rest of the algorithm is executed in O(1) time.

8.3 The first edge-colouring algorithm

Algorithm 8.2
Complete binary splitting (Edge colouring)

input: the matrices V and E, d

!each proc par-do 1
  D <-$exp_2(\lceil \log_2 d \rceil)$

```
for i=0 step 1 until ⌈log2d⌉-1 do 2:
    Call graph-partition (algorithm 8.1)
    input [V,E,D]

    D <- D/2
end do 2

(The contents of E¯OT(*) gives a consistent colour assignment.)

end par-do 1
end of procedure.
```

Theorem 8.2

Let $D=exp_2(\lceil \log_2 d \rceil)$. Algorithm 8.2 finds a D edge colouring in $O(\log d \cdot \log E)$ time. For a graph whose degree is a power of 2 this is a minimal edge colouring.

Proof

Each iteration of the loop do-2 calls graph-partition, which requires $O(\log E)$ time. At the beginning of the algorithm the degree of each node does not exceed D. It is easy to prove by induction that after the t'th iteration of the algorithm the degree of each node does not exceed $D/(2^t)$. Hence after $\lceil \log_2 d \rceil$ iterations the degree of each node does not exceed 1, and each subgraph can be coloured in one colour.

8.4 The second edge colouring algorithm, using recolouring

In the second edge colouring algorithm the graph is split into two graphs, and each half is coloured (recursively) in parallel, to give a $2\lceil d/2 \rceil$ colouring to G. If $2 \cdot \lceil d/2 \rceil > d$, i.e. the two half graphs are coloured in d+1 colours, then the edges of one colour group, say, the edges coloured d+1, are recoloured. The

recolouring procedure uses an alternating path method, in which the paths are chosen so that a large number of edges may be recoloured in parallel.

A noncoloured edge may be typed or untyped. If it is typed, a type (tentative colour) is associated with each edge-end. The edge is designated as being of (i,j) type if at one end (the one typed i) it is not adjacent to any edge coloured i or to any edge-end typed i, and at the other end it is not adjacent to any edge coloured j or to any edge-end typed j.

An (i,j) coloured path is a path of alternating i and j coloured edges.
If no two noncoloured edges in the graph are adjacent to each other, then an (i,j) coloured path can have at each end at most one (i,j) type edge.
An (i,j) path is a maximal path of i and j coloured edges and of (i,j) type noncoloured edges. It is defined only in bipartite graphs without adjacent noncoloured edges.

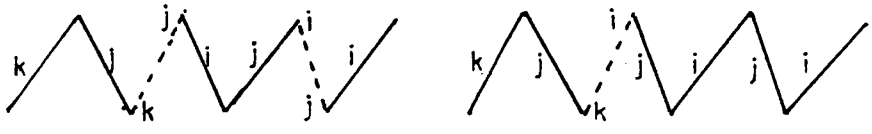
Lemma 8.2

Suppose that in G every noncoloured edge is adjacent only to coloured edges, and let each noncoloured edge be assigned a type. For fixed i,j let all the (i,j) paths be parity labelled and recoloured: edges labelled 0 are coloured i and edges labelled 1 are coloured j. The recolouring is performed so that in each path at least one coloured edge retains its original colour. Then the number of noncoloured edges which change their type as a result of the recolouring is at most the number of noncoloured edges which become coloured.

Proof

A typed edge may have to change its type as a result of the recolouring only if it contains i or j (but not both) in its type.

By the assumption such a non coloured edge is adjacent only to coloured edges, and therefore it is adjacent to an edge coloured i or j which changed its colour, at the end of an (i ,j) path. At least one edge on the (i,j) path retained its colour. Thus on the path from the coloured edge which changed its colour to the coloured edge which retained it, there is an (i,j) type edge which got coloured. For each edge which changed its type there is at least one (i,j) type edge which got coloured.



before recolouring after recolouring

Figure 8.1

Lemma 8.3

Let G be coloured in d colours, and let all the noncoloured edges of G be nonadjacent. Let the colours be indexed from 0 to d-1.

a) For every $g \in \{0,1,\ldots,d-1\}$ the set of all the (i,j) paths which satisfy $i+j=g \bmod d$, $g=0,1,\ldots d-1$ may be recoloured in parallel.

b) If for every $g \in \{0,1,\ldots d-1\}$ the paths which satisfy $i+j=g \bmod d$ are recoloured (in parallel), then at least half the noncoloured edges are recoloured.

c) If G has n input nodes, after at most $\log_2 n$ iteration of recolouring (in parallel, as above) for every $g \in \{0,1,2,\ldots,d-1\}$, all the noncoloured edges are coloured.

Proof

a) For $g=0,1\ldots d-1$, two paths in the same group do not have common colours.

b) Recolouring all the edges satisfying $i+j=g \bmod d$ for $g=0,1...d-1$, all the colour pairs get recoloured. The edges which remain uncoloured are only those which changed their type. By lemma 8.2 for each edge which changed its type at least one edge got coloured. Therefore at least half the noncoloured edges are coloured.

c) By b), at each iteration at least half the noncoloured edges are coloured. By the asumption of the theorem at most one noncoloured edge is incident with each node, therefore there are at most n noncoloured edges, and after $\log_2 n$ iterations all of them get coloured.

Algorithm 8.3 calls complete binary splitting which splits the graph (in parallel) to $\exp_2\lceil \log_2 d \rceil$ subgraphs. The pointers to edges belonging to the same subgraph are stored in one column of the nodes' matrix. Throughout the rest of the algorithm, each column of the nodes' matrix represents one colour. The two pointers to a coloured edge are stored in the same column, and pointers to typed edges are stored in the columns representing their types. Pairs of subgraphs with degree $\lceil d'/2 \rceil$ are combined (in parallel) to subgraphs of degree $2\cdot\lceil d'/2 \rceil$. If $2\cdot\lceil d'/2 \rceil = d'+1$, then the edges indexed d' are assumed noncoloured. They are typed using types 0 to d'-1. (d' types altogether). There is at most one noncoloured edge incident with each node, as the noncoloured edges represented a colour in a d'+1 colouring of the subgraph. For every $g \in \{0,1,...,d'-1\}$, (i,j)-paths are created for edges which satisfy $i+j=g \bmod d'$ (in parallel). Parity 0 is associated with the smallest indexed colour in each pair i,j, defining an (i,j) path. The edges get parities compatible with their colours, and coloured edges get higher priority than typed edges. The paths are parity labelled and recoloured. The priority ensures that at least one previously-coloured edge retains its colour. This is repeated $\log_2 N$ times until all the noncoloured edges get coloured.

Algorithm 8.3

Second parallel edge colouring using recolouring

Let $D=\exp_2\lceil \log_2 d \rceil$

input The matrices V[0:N-1,0:D] and E. The value d.

Call complete binary splitting (algorithm 8.2)

input [V ($V^{-}D$ <- d), E, D]

(main-loop)

for $i=\lceil \log_2 d \rceil - 2$ step -1 until 0 do 1

Assign one processor to each position in the matrix V.

!each proc par-do 2

$j \leftarrow a_D(*)$; $k \leftarrow b_D(*)$

$D' \leftarrow D/2^i$; $d' \leftarrow \lceil d/2^i \rceil$

$k_a \leftarrow a_{D'}(k)$; $k_b \leftarrow b_{D'}(k)$

(k_b is the position in the list of subgraph k_a.)

combine pairs of subgraphs to one subgraph by storing it in consecutive columns of V.

$L \leftarrow D'/2 - \lceil d'/2 \rceil$

if $a_2(k_b)=1$ then do 3

$V(j,k-L) \leftarrow V(j,k)$

$E^{-}OT(V^{-}PT(j,k),V^{-}OT(j,k)) \leftarrow k-L$ (through local memory)

if $k_b - 2\cdot\lceil d'/2 \rceil > 0$ then $V(j,k) \leftarrow \phi$

end do 3

end combine subgraphs.

if $2\cdot\lceil d/(2^{i+1}) \rceil > \lceil d/(2^i) \rceil$ then do 4

(typed recolour)

(the edges indexed d' are assumed non coloured.)

(Assign type to noncoloured edges)

(Assign the first missing colour in each subgraph to

the noncoloured edge end by storing the edge-end data block in the corresponding column).

Find the index of the first missing colour in the list of each subgraph in the row of each node, store its value in V⁻RS. (using variation of algorithms 7.4, 7.3)

```
if k_b=d' and V(j,k)≠φ then do 5
    E⁻OT(V⁻PT(j,k),V⁻OT(j,k)) <- V⁻RS(j,k)
    V(j,V⁻RS(j,k)) <- V(j,k); V(j,k) <- φ
end do 5
(end assign type)
```

(noncoloured edge is pointing to different columns corresponding to different colours).

```
(recolour loops)
for q=1 step 1 until log2N do 6
    for g=0 step 1 until d'-1 do 7
```

(create (i',j') paths)
(A coloured edge and typed edges whose types (i',j') satisfy i'+j'=g mod d' are participating in paths. Edges whose coloums satisfy i'+j'=g mod d' are made adjacent)

```
if V(j,k)≠φ then do 8
    t <- (g-k_b)mod d'
    t_0 <- min(k_b,t)     t_1 <- max(k_b,t)
    n <- V⁻PT(j,k)        m <- V⁻OT(j,k)
    V⁻TG(j,k) <- φ
    E⁻TG(n,m) <- φ
    if b_d'(E⁻OT(n,m̄))=t or E⁻OT(n,m̄)=k then do 9
        V⁻TG(j,k) <- 1
        E⁻TG(n,m) <- 1
        t' <- p_D,(k_a,t)
        if V⁻TG(j,t')=1 then do 10
            E⁻L⁻PT(n,m) <- V⁻PT(j,t')
            E⁻L⁻OT(n,m) <- V⁻OT(j,t')
```

```
                end do 10 else E⁻L⁻PT(n,m) <- ϕ
                E⁻L⁻AR(n,m) <- n
                if E⁻OT(n,m)=E⁻OT(n,m̄) then
                    E⁻L⁻AR(n,m) <- E⁻L⁻AR(n,m) +|E|
                if k_b ≤ t_0 then E⁻L⁻IP(n,m) <- 0
                else E⁻L⁻IP(n,m) <- 1.
            end do 9
        end do 8
        (end create paths)

        Call Parity labelling (algorithm 7.1)
        Input E⁻L as the list
        (The maximal length of each list is 2N)

        (store coloured edges)
        if E⁻L⁻IP(n,m)=0 or E⁻L⁻PT(n,m)=ϕ then do 11
            t_0' <- p_D'(k_a,t_0)
            t_1' <-p_D'(k_a,t_1)
            Copy to local memory V(j,k), V(j,t').
            if E⁻L⁻IP(n,m)=0 then do 12
                E⁻OT(n,m) <- t_0'
                E⁻OT(v⁻pt(j,t'),v⁻ot(j,t')) <- t_1'
                V(j,t_0') <- v(j,k)
                V(j,t_1') <-v (j,t')
            end do 12 else do 13
                E⁻OT(n,m) <- t_1'
                E⁻OT(v⁻pt(j,t'),v⁻ot(j,t')) <- t_0'
                V(j,t_1') <- v(j,k)
                V(j,t_0') <- v(j,t')
            end do 13
        end do 11
        (end store coloured edges) (the storing assigns
        types to the remaining noncoloured edges.)
    end do 7
end do 6
(end recolour loops)
```

(end typed recolour)
(end do 4)

(end par-do 2)

(end do 1)
(end main loop)

(end of procedure)

Theorem 8.3

For each graph whose degree $d=2^k \cdot C$, algorithm 8.3 finds an edge colouring in $O(C \cdot \log^2 N + \log E \cdot \log d)$ time $= O(C \cdot \log^2 N)$ time and $O(N \cdot d)$ space.

Proof

Space.
The matrices V,E require $O(N \cdot d)$ space.

Time.
Algorithm 8.3 calls algorithm 8.2, which requires $O(\log d \cdot \log E)$ time.
Loop 1 is executed $\log_2 d$ times.
At each execution of loop 1 combining pairs of subgraphs requires constant time, adding altogether $O(\log d)$ time.

If $d=2^k \cdot C$, then for i=0 to k-1
$2 \cdot \lceil d/2^{i+1} \rceil = \lceil d/2^i \rceil$,
and typed recolour is not executed.
Typed recolour is called only for $i=\lceil \log_2 d \rceil - 2$ up to i=k.
In each execution of typed recolour
Assign types requires $O(\log d)$ time.
Loop 6 is executed $\log_2 N$ times

and loop 7 is repeated O(d') time.

Each execution of loop 7 calls Parity labelling on lists of edges of length 2N (at most).

Therefore each execution of Typed recolour requires $O(d' \log^2 N)$ time.

Substituting $d' = \lceil d/2^i \rceil$,

The time spent on recolouring is

$$\sum_{i=\lceil \log_2 d \rceil}^{k} C' \frac{d}{2^i} (\text{const}) \log^2 N$$

$$= \text{const} \cdot \log^2 N \cdot (d/2^k) \sum 1/2^i$$

$$= O(\log^2 N \cdot C) \text{ time.}$$

At each iteration of loop 6 all the combinations of colour pairs get recoloured, and the non coloured edges which are not recoloured are edges which changed their type. By lemma 8.3 after $\log_2 N$ iterations of loop 6 all the noncoloured edges get coloured.

8.5 The third edge colouring algorithm.

This algorithm is faster than the second algorithm for dense graphs. Only in very sparse graphs, d being a small constant, does the second algorithm fare better than the third one.

Let @ be the maximal power of 2 less than or equal d. $@ = \exp_2 \lfloor \log_2 d \rfloor$. The algorithm assigns to each noncoloured edge a type so that a colour occurs at each node at most once. (A colour i occurs at a node if an incident edge is coloured i, or an incident edge-end is typed i). Then, a set S of @ colours which contains a large proportion of the colour pairs assigned to typed edges is picked. The set of all edges that either have colours from S or have both types from S, form a subgraph of degree @ which can be recoloured quickly using algorithm 8.2. This colours

all the typed edges which had both their types in the set S. This is repeated until the graph is coloured.

The termination condition relies on the fact that at each iteration a large proportion of the noncoloured edges are picked and get coloured. Using the sequential algorithm (Gabow and Kariv (to appear)), it is possible to guarantee that 1/4 of the noncoloured edges get coloured at each iteration, and at most $\log_{4/3}E$ iterations are required. We define z and y by:

$@=z\cdot d, \quad 1/2 < z \leq 1$

and $d-@=y\cdot d \quad 0 \leq y < 1/2$.

Their sequential algorithm guarantees, in fact, that at least $z^2-(1-z)z/(d-1)\approx z^2$ edges get coloured in each iteration. If sorting is used in the parallel implementation as we shall here, it is possible to guarantee that at least $(1-y)(1-(2y/(2-y)))$ $\geq 1/6$ of the noncoloured edges get coloured in each iteration. Without sorting, it is possible to guarantee that at least 1/6 of the noncoloured edges get coloured in each iteration. The last possibility, suggested by Pippenger, is presented in Lev-Pippenger-Valiant(to appear).

Theorem 8.4

Let $d-@=yd$. Assume that all the non coloured edges are typed, and the colours for recolouring are chosen as follows:

a) The number of typed edge ends assigned to each colour is counted.
b) The lowest $y\cdot d/2$ ranking colours are "discarded" and all the edges coloured or typed (in one of their ends) in these colours are discarded as well.
c) In the remaining graph the number of typed edge ends assigned to each colour is counted.
d) The lowest $y\cdot d/2$ ranking colours among those are "deleted" as well.

At least $(1-y)(1-(2y/(2-y)))$ of the typed edges are chosen to be

coloured in this way.

Proof

Let e be the number of non coloured edges.

The $y \cdot d/2$ lowest ranking colour groups contain at most $(y \cdot d/2) \cdot (2e/d)$ edge ends. When the edges incident with them are erased, at most $y \cdot e$ typed edges are erased, and at least $e(1-y)$ typed edges are left. The lowest ranking $y \cdot d/2$ colours in this group contain at most $(y \cdot d/2) \cdot ((2e(1-y)/(d-yd/2))$ typed edge ends, which reduce the number of edges by at most that much. The number of typed edges left for recolouring is at least
$(1-y)e - (yd/2)(2e(1-y)/(d-yd/2)) =$
$(1-y)(1-2y/(2-y))e$ edges.
This is a decreasing function of y, and for $y=1/2$ at least 1/6 of the typed edges get coloured.

Algorithm 8.4 (outline)
Parallel edge colouring using colouring by pairs

Input: G represented by Neil matrices, [V[0:n-1,0:d-1], E]

!each proc par do
  g <- (1-y)(1-2y/(2-y))
  f <- 1/(1-g)
  Copy the matrix V to V' and E to E'.

  for i=0 step 1 until $\log_f E$ do 1
    Assign types to uncoloured edges.
    (the matrix position defines the type. The ends of typed edges are tagged)

    Count the number of typed edges-ends assigned to each colour.
    (the counting is performed on the columns of the matrix V'

considered as lists.)

Sort the colours according to the number of typed edge-ends assigned to them.
(It is possible to perform bucket sort: if j' edge-ends are coloured k', the information is stored in position (j',k') in an empty field in the matrix V'. Counting the number of colours with the same value of j' requires O(log N) time, and finding the sorted position of each group of colours with the same value of j' requires O(log d) time. Transferring the information to each colour requires additional O(log N+log d) time.)

Erase all the lowest yd/2 ranking colours and the edges incident with them.
(Insert ϕ in all the columns corresponding to those colours.)

Count the number of the remaining typed edge-ends assigned to each colour.

Sort the colours according to the number of typed edge-ends assigned to them.

Erase the lowest yd/2 ranking colours and the edges incident with them.

The remaining graph has degree @, a power of 2. Colour all the edges of the chosen colours, using algorithm 8.2.
(The index of each non-deleted colour in the matrix V' is counted. The active columns are stored in the first @ columns, and algoritm 8.2 is called. At its termination, column k is stored in the column corresponding to the k'th active colour, in the matrix V (updating E as well).)

end do 1

end par-do.
end of procedure

Theorem 8.5

Algorithm 8.4 finds an edge colouring in $O(\log^2 E \cdot \log d)$ time and $O(N \cdot d)$ space.

Proof

The space requirements are obvious.
Each iteration of loop 1:
Performs sorting and counting which requires $O(\log N + \log d) = O(\log E)$ time.
It calls algorithm 8.2 which requires $O(\log E \cdot \log d)$ time.
The loop is executed $O(\log E)$ times.

Each execution of the algorithm's loop eliminates at least g ($\geq 1/6$) of the noncoloured edges, and in $\log_f E$ ($\leq \log_{6/5} E$) iterations of the loop all the edges of the graph get coloured. Therefore the algorithm requires $O(\log^2 E \cdot \log d)$ time.

We saw that if d is a power of 2, edge colouring is performed by algorithm 8.2, which is executed only once, and requires only $O(\log E \cdot \log d)$ time. This algorithm is executed only once even if algorithm 8.3 or 8.4 were called in the first place.

If $d = C \cdot 2^k$, algorithm 8.3 requires $O_k(C \cdot \log^2 N)$ time. If C is very small compared with E, then algoritm 8.3 is competitive with algorithm 8.4.

For dense graphs, when the degree of the graph is not a multiple of a large power of 2, algorithm 8.4 gives the best results.

9 PARALLEL ALGORITHM FOR ROUTING IN PERMUTATION NETWORKS AND SUPERCONCENTRATORS

9.1 Introduction

This short chapter brings together the constructions and the algorithms described earlier. Given a PN or a SC, constructed recursively from smaller size networks of the same kind, and given a connection requirement $\Sigma = \{(i, c(i))\}$, we seek to find a routing by node disjoint paths from the active input nodes to the active output nodes. The proofs that recursively constructed PNs and SCs are indeed networks with the properties ascribed to them (theorem 2.1, theorem 2.2) provide routing algorithms.

If the PNs, SCs are constructed uniformly from networks of size d, then the parallel routing algorithm finds a routing in time:

$$\min \begin{cases} O_d(\log_2^2 n) & \text{if } d \text{ is a power of 2 } (c=1) \\ O(c(\log_2 c + k)\log_2^3 n) & d = 2^k \cdot c \\ O_d(\log_2^3 n) & \end{cases}$$

The routing algorithms call for an edge colouring algorithm for each step of the recursive construction. The edge colouring algorithms for bipartite graphs described earlier, did not assume knowledge of the groups of the bipartite graph, they only assumed that the graph is bipartite. In the routing algorithm these groups are known: one corresponds to the input terminals of the network and the other to the output terminals. It is possible to take advantage of this knowledge by specializing the edge colouring algorithm and using half the number of processors. We did not make this assumption in the following algorithm. It is implemented in Lev-Pippenger-Valiant (1980).

9.2 Routing in Permutation Networkss

In constructing the σ routing algorithm we assume that the structure of the network is known to each processor. The parameters known to each processor include:

(i) n - the number of input and output nodes in the network. (Or n_i, n_o in case they are different).

(ii) depth - the network's depth.

(iii) struct[0:depth-1] - struct(i) is the size of the subnetworks A,C in (A,B,C)-A at the i'th iteration of the recursion.

(vi) miss[0:depth-1] - miss=1 when the node is an input node in a missing copy of A in (A,B,C)-A.

When the network is constructed from subnetworks of the same size d, only the value of d is required, instead of depth and struct. When the network is constructed from subnetworks of size d and d-1 more parameters are needed, as discussed later on.

Algorithm 9.1
Parallel algorithm for routing in PNs.

Input: [CC[0:n-1]]
output [CN[0:2n-1,0:depth]]

(CC(i) specifies the permutation requirement, CC(i)=c(i). When Σ is a partial permutation CC(i)=ϕ for nodes not participating in the permutation. CN(i,t) gives the colour assigned to node i i.e. the output terminal assigned to b(i) in the subnetwork, at the t'th iteration of the construction.)

```
!each proc par do
   N <- 2n

   if * < n then do 1
   (Initiation)
       CC'(*) <- CC(*)
```

(end initiation)
end do 1

(Main-loop)
for t=0 step 1 until depth-1 do 2

1) Let CC' contain the permutaion assignment. Input node i is associated with output node CC'(i). If it is idle then CC'(i)=ϕ.

d <- struct(t); j <- $a_d(*)$; k <- $b_d(*)$

if * < n then do 3
DD'(*) <- ϕ
if CC'(*)$\neq\phi$ then DD'(CC'(*)) <- *

2) Construct a bipartite graph with an edge between each pair of associated nodes.

(Keeping CC', DD' is equivalent to keeping the following Nal representation of the bipartite graph:
V'(*) <- CC'(*)+n V'(CC'(*)+n) <- *)

3) Construct G' from G by identifying all the input (output) nodes with the same value of a(i). a(i) becomes the node index, b(i) the orientation.

(In constructing the bipartite graph, if i is the r'th active input node, then (i,c(i)) is the r'th edge. Input nodes have orientation 0 in the edges' list and output nodes have orientaion 1.)

V⁻PT(j,k) <- ϕ; V⁻PT(j+n,k) <- ϕ
E⁻PT(*,0) <- ϕ; E⁻PT(*,1) <- ϕ
r <- the index of non-empty CC'(*) in CC'( ) considered as a list. (r=r(*))

if CC'(*)$\neq\phi$ then do 4

V⁻PT(j,k) <- r; V⁻OT(j,k) <- 0

V⁻N(j,k) <- *

E⁻PT(r,0) <- j; E⁻OT(r,0) <- k

j' <- a_d(CC'(*)); k' <- b_d(CC'(*))

V⁻PT(n+j',k') <- r; V⁻OT(n+j',k') <- 1

V⁻N(n+j',k') <- CC'(*)+n

E⁻PT(r,1) <- n+j'; E⁻OT(r,1) <- k'

(⁻N keeps the value of the original index of the node. After the edge colouring algorithm, the orientation will represent the colour assigned to the node.)

end do 4

end do 3

4) Find an edge colouring in which the edges incident with the missing copy of A are coloured b(i).

Call edge-colouring (algorithm 8.3 or 8.4)

Input [V[0:(N/d)-1, 0:D], E, d]

j <- a_d(*); k <- b_d(*)

(colour-permutation)

(find-permutation)

Find a complete permutation of the colours, so that the edges (i,c(i)) incident with the missing copy of A get coloured b_d(i), store the result in PERMUTE[0:d-1]. PERMUTE(i) is the permutation of colour i.

(end find-permutation)

Propagate the value of PERMUTE(i) to column i.

E⁻OT(V⁻PT(j,k),V⁻OT(j,k)) <- PERMUTE(j,k)

V(j,PERMUTE(j,k)) <- V(j,k)

(colour permutation is ignored if the network is constructed as (A,B,C).)

(end colour-permutation)

5) Store the connection information and establish the connection requirement for the next iteration.

$CN(\bar{V}N(k,j),t) \leftarrow j$

if $k < n$ then do 6

$CN_1(\bar{V}N(k,j)) \leftarrow p_d(j,k)$

$NC_1((\bar{V}N(k,j) \leftarrow p_{n/d}(j,k)$ end do 6

else $(k \geq n)$ do 7

$CN_2(\bar{V}N(k-n,j) \leftarrow p_d(k-n,j)$

$NC_2(\bar{V}N(k,j) \leftarrow p_{n/d}(j,k-n)$

end do 7

Output node CN_1 (in a subnetwork) is identified with input node NC_1 in the following internal stage. Input node CN_2 is identified with ouput node NC_2 in an internal stage of the construction. Now, for $i<n$ the connection requirement for the next iteration is established.)

if $* < n$ then do 8

$CC'(NC_1(*) \leftarrow NC_2(CC'(*))$

end do 8

end do 2

(end main loop)

end par do

end of procedure.

Theorem 9.1

Assume that a PN is built recursively from subnetworks of size d. If $d=2^k \cdot c$. then algorithm 9.1 finds a routing in the network in time:

$$\min \begin{cases} O_d(\log_2^2 n) & \text{if d is a power of 2 } (c=1) \\ O(c/(k+\log c)\log_2^3 n) & \\ O_d(\log_2^3 n) & \end{cases}$$

Proof

Each iteration of the main loop in algorithm 9.1 calls edge colouring on a graph of size 2n/d nodes and at most n edges.

If d is a power of 2, each call to the edge colouring algorithm calls algorithm 8.2, and requires $O(\log_2 d \cdot \log_2 n)$ time. The main loop is iterated $\log_d n$ times, so the total time required is $O(\log_d n \cdot \log_2 d \cdot \log_2 n) = O(\log_2{}^2 n)$ time.

If $d=c\cdot 2^k$, and algorithm 8.3 is called for edge colouring, each call for edge colouring requires $O(c\cdot \log_2{}^2 n)$ time. Since this is repeated $\log_d n$ times, the total time required is $O(c\cdot \log_d n \cdot \log_2{}^2 n)$ time.

If algorithm 8.4 is called for edge colouring each iteration of main-loop takes $O(\log_f n \cdot \log_2 n \cdot \log_2 d)$ time, and the total time for executing algorithm 9.1 is $O(\log_f n \cdot \log_2 n \cdot \log_2 d \cdot \log_d n) = O(\log_f n \cdot \log_2{}^2 n) \leq O(\log^3 n)$ time.

9.3 Routing in Superconcentrators

For SCs the routing algorithm finds only the active input and output nodes in each subnetwork. Assuming d to be very small, the routing in each subnetwork of size d can be found, using a sequential flow algorithm, in parallel for all the subnetworks at the same stage of the construction.

Assume that the connection requirement is stored in two arrays I and O, specifying the indices of the active input and output nodes respectively. The number of active input (output) nodes: n' is given also. Before starting the algorithm we need to create an explicit assignment from the requirement.

Algorithm 9.2
Routing in SCs

input [n',I[0:n'-1],O[0:n'-1]]

Execute algorithm 9.1 with the following modification:
Add before initiation
!each proc < n par do
   CC(*) <- ϕ
   if * < n' then CC(*) <- O(*)
end par do

If the SC was constructed as in case b of theorem 2.2, at each stage of the construction we need to keep, for example, the first node in the subnetwork (frs(*)) and the value of q in the subnetwork. (see chapter 2, definition of $a_{d,q}(i)$). Each occurrence of $a_d(i)$ in the algorithm now becomes $a_{d,q}(i-frs)$, and similarly for b and p.

In this case the assignment should be an order preserving assignment. In addition all the edges incident with C' need to be coloured in the first s indexed colours.

Algorithm 9.2'
Routing in SC, with order preserving assignments.

Execute algorithm 9.2 with the following modifications:

Add at the begining of step 1 (begining of loop-2)
Create temporary arrays ITMP[0:n-1],OTMP[0:n-1]
!each proc par do
   ITMP(*) <- ϕ; OTMP(*) <- ϕ
   ITMP(*) <- (if CC'(*)=ϕ then 0 else 1)
   OTMP(*) <- (if DD'(*)=ϕ then 0 else 1)
   Find for ITMP,OTMP the weighted position of each item in the list and store in IRS,ORS respectively.

```
   if ITMP(*)≠0 then ITMP(IRS(*)) <- *
   if OTMP(*)≠0 then OTMP(ors(*)) <- *
   CC'(*) <- ϕ;   DD'(*) <- ϕ
   if * < n' Then CC'(ITMP(*)) <- OTMP(*)
end par do.
```

Adjust find-permutation to ensure that all the edges (i,c(i)) incident with C' are assigned colours less than s.

end

Some remarks on colour-permutation.

When the inital assignment is a complete permutation, then the following routine finds the required permutation:

```
(find-permutation)
   if miss(j,k)=1 then PERMUTE(k) <- b_j(V⁻N(j,k))
(end find-permutation)
```

If the assignment is not a complete permutation it can be increased to a complete one, or, alternatively, the colour permutation at find-permutation can be enlarged.

When the network is a SC, built recursively as in case b of theorem 2.2, then a more complicated routine can find the permutation. A possible solution is to create two vectors, say S,T, of size d, such that the colour S(i) is permuted to T(i), and then to store PERMUTE(S(i)) <- T(i). The structure of S and T can be described as follows

T	For each i $b_d(V^{-}N(S(i))$	colours not appearing on the left, sorted. (No sorting for PNs)	
S	colours assigned to active nodes in A'(in the missing copy of A)	colours assigned to nodes in C' which are not adjacent to A'	colours not appearing on the left

The vectors S,T can be easily found with the help of an auxiliary vector, and a routine which packs to the left all the non empty entries in the vector and counts them.

10 PARALLEL ALGORITHM FOR MAXIMAL MATCHING IN BIPARTITE GRAPHS

10.1 Introduction and definitions

Assume the following restricted model of a network: there is a set of input terminals connected to a set of output terminals. Each terminal can implement one "request" at a time, yet many requests might be issued for each terminal concurrently. We represent the requests by a bipartite graph, and each "request" as an edge between an input node and an output node. The best usage of the system will be obtained when a maximum matching can be implemented by connections. In some cases, however, we may be satisfied with any reasonable large matching. In this chapter we consider one such nonoptimal requirement, namely maximal matching.

A matching M is a subset of the edges of the graph, such that any two edges in M are node disjoint. The cardinality of the matching M is the number of edges in the matching.

A maximal matching is a matching to which it is impossible to add another edge from G which does not have a common node with M.

A maximum matching is a maximum-cardinality matching.

A node which is not incident with any matched edge is a free node.

The best parallel maximum matching algorithm known to me requires O(N log N log d) time and E+N processors. It is not presented here. (Compare with the best sequential algorithm of Hopcroft and Karp (1973), which requires $O(N^{5/2})$ time). The best sequential algorithm for maximal matching requires O(E) time. Below we shall give three parallel algorithms for maximal matching, each using a modest amount of space. For graphs with N

input nodes, M output nodes and degree d they give an upper bound on parallel time complexity of

$$O\left(\min\begin{cases} d\cdot\log d \\ (N+M)^{1/2}\cdot\log E\cdot\log d \\ \log^2 d\cdot\log N\cdot\log E \end{cases}\right)$$

Each bound corresponds to an algorithm. The first one is a parallel implementation of the sequential algorithm, the other two use complete binary splitting (algorithm 8.2) as a subroutine.

In the special case when the graph is regular or semiregular (defined later), a maximum matching can be found quickly.

10.2 Special case: regular and semiregular graphs

A regular graph is a graph in which the degree of all the nodes equals d.

A semiregular graph is a bipartite graph in which the degree of all the nodes in one group, say A, equals the maximal degree d.

Using the edge colouring algorithms it is possible to find a maximum matching in these graphs in at most $O(\log^2 d\cdot\log E)$ time.

Theorem 10.1

In a regular or semiregular graph with degree d it is possible to find a maximum matching in:

$$\min\begin{cases} O(\log_2 d\cdot\log_2 E) \text{ time} & \text{if d is a power of 2} \\ O_k(c\cdot\log_2{}^2 N) \text{ time} & \text{if } d=c\cdot 2^k \\ O(\log^2 E\cdot\log_2 d) \text{ time.} \end{cases}$$

Proof

In a regular or semiregular graph each colour in a minimal edge colouring is a maximum matching. In a minimal edge colouring the

graph is coloured in d colours. Each node in one of the node-groups of the bipartite graph has degree d, therefore each such node is incident with one edge from each colour group. Each colour group is incident with all the nodes in this group of the bipartite graph (in the two groups if the graph is regular) and the matching cannot be extended. Therefore the minimal time required to find a maximum matching in this case, is at most the minimal time to find an edge colouring, cited above.

Let G be a bipartite graph with N,N' nodes in its two groups of nodes, A,B, respectively and E edges. Assume $N \leq N'$. Let d(A),d(B), be the maximal degrees of nodes in A,B. A maximum matching contains at most N edges, using the edge colouring algorithm it is possible to find a matching with $\lceil E/d(A) \rceil$ edges. In "dense" graphs, i.e when $\lceil E/d(A) \rceil$ is almost N, this gives a good approximation to maximum matching.

10.3 First algorithm for maximal matching in bipartite graphs.

This algorithm is a parallel implementation of the sequential one. The sequential algorithm requires O(E) time. This algorithm requires O(d·log d) time. Let G(A,B) be a bipartite graph with N_A, N_B nodes in A,B respectively. Let $N=\max(N_A,N_B)$, $d(A)=\min(d(A),d(B))$, and $D=2^{\lceil \log_2 d \rceil}$. This algorithm needs to know the two groups of nodes A,B. Assume that the graph is represented by a 3 dimensional oriented Val matrix V[0:1 , 0:N-1 , 0:D-1], where index 0 in the additional dimension identifies nodes in A, and index 1 identifies nodes in B. It is implicit in this description that pointers from A point to B and vice versa.

Algorithm 10.1

input [V[0:1,0:N-1,0:D-1],d]

```
!each proc par do
   Let j=a_d(*), k=b_d(*)

   initiation
   for i=0,1 do 1
       V⁻MT(i,j,k) <- 0    (the matching is empty)
       V⁻IN(i,j,k) <- 1    (data blocks are tagged as active)
   end do 1
   end initiation

   for t=0 step 1 until d-1 do 2

       1)  Choose one edge incident with each node in A as a
       candidate for matching.
       For each j find a k', say, with the highest priority, such
       that     V⁻IN(1,V⁻PT(0,j,k),V⁻OT(0,j,k))=1 and V⁻IN(0,j,k)=1
       V⁻MT(1,V⁻PT(0,j,k'),V⁻OT(0,j,k')) <- 1
       (V⁻MT contains candidates for matching in B and matched
       edges in A).

       2)  For each node in B choose one of the candidates for
       matching incident with it (if one exists), and add it to
       the matching.
       For each j choose one k, say k' such that V⁻MT(1,j,k')=1
       and V⁻IN(1,j,k')=1
       V⁻MT(0,V-PT(1,j,k'),V⁻OT(1,j,k')) <- 1

       3)  Erase from G all the edges incident with nodes which
       are at ends of edges just added to the matching.
       V⁻IN(1,j,k') <- 0
       V⁻IN(0,V⁻PT(1,j,k'),V⁻OT(1,j,k')) <- 0
       for i=0,1 do 3
           V⁻IN(i,j,k) <- min_k(V⁻IN(i,j,k))
       end do 3
   end do 2
end par-do
```

end of procedure.

Theorem 10.2

Algorithm 10.1 finds a maximal matching in $O(d \cdot \log_2 d$ time).

Proof

Each iteration of loop 2 requires $\log_2 d$ time, as each of the operations in steps 1 to 3 is performed on the rows of the matrix and can be done in $O(\log_2 d)$ time. A free node in A can remain active and with (active) degree greater than zero for at most d iterations. At each iteration of loop-2 a node in the group with degree d is either matched, or its degree is reduced by one, as a node adjacent to it stops being free. After d iterations there are no more free nodes which are adjacent to free nodes in the graph and the algorithm terminates. The total time required is $O(d \cdot \log_2 d)$.

10.4 Second maximal matching algorithm for bipartite graphs.

In this algorithm the knowledge of the groups A,B is not required. The graph is assumed to be represented by a Neil, and most of the operations are performed on the edges. The algorithm calls for algorithm 8.2 as a subroutine. Let G(A,B) be a bipartite graph with $N=N_A+N_B$ nodes and E edges.

Algorithm 10.2
input [V[0:N-1],E[0:E-1]]

initiation
Set all the nodes and edges in the graph as active.
Set the nodes and edges as unmarked.
end initiation

procedure 1

1) Count the degree, d(v), of each (active) node in the graph.
   Find the degree of the graph, d.
   if d=0 then terminate, a maximal matching has been found.

2) Count the number of active nodes N in G.
   Count the number of active edges E in the graph.

3) Mark each edge that is incident with at least one node whose degree is at least $E/(N^{1/2})$
   Count the number of marked edges and store in E'.
   (i.e. count the number of edges, E' incident with nodes whose degree d(v) is at least $E/(N^{1/2})$, an edge incident with two such nodes being counted once.)

4) if E'>E/2 then do 2

   (sequential operation)
      Index the nodes whose degree is at least $E/(N^{1/2})$
      for i=0 step 1 until E'-1 do 4
         If the node indexed i is active, select an edge incident with it and add it to the matching.
         Make this edge, the nodes incident with it, and the edges adjacent to it inactive.
      end do 4
      call procedure 1
   (end sequential operation)

   end do 2 else do 3

   (parallel operation)
      Consider each marked edge as inactive and
      call complete binary splitting (algorithm 8.2) on the graph composed of active unmarked edges. (The complete binary splitting splits this graph into at most

$\exp_2(\lceil \log_2(E/N^{1/2}) \rceil)$ subgraphs.)
Find the subgraph (in the splitting) with maximum number of edges.
Add all the edges in this subgraph to the matching
Deactivate all the nodes incident with the edges added to the matching and all the edges incident with those nodes.
Erase all the marks.
call procedure 1
(end parallel operation)

end do 3
end of procedure 1
end of algorithm.

Theorem 10.3

Algorithm 10.2 finds a maximal matching in at most $O((N)^{1/2} \cdot \log_2 E \cdot \log_2 d)$ time.

Proof

Each time procedure 1 is called (recursively), steps 1,2,3 are executed in $O(\log_2 d + \log_2 N + \log_2 E) = O(\log_2 E)$ time.

Sequential execution requires $O(N^{1/2} \cdot \log_2 d)$ time until it calls procedure 1 again: There are at most $(N)^{1/2}$ nodes whose degree is at least $E/(N^{1/2})$, and for each such node choosing an edge and erasing all the edges adjacent to it and the nodes incident with it requires $O(\log_2 d)$ time.
Indexing the nodes requires at most $O(\log_2 E)$ time.

Parallel operation requires $O(\log_2 E \cdot \log_2 d)$ time before it calls procedure-1. Each execution of parallel-operation calls complete-binary-splitting, which requires $O(\log_2 E \cdot \log_2 d)$ time.
All the other operations require at most $O(\log_2 E)$ time.

As a result of sequential operation at least half the edges are eliminated.

As a result of parallel opereation at least $(N^{1/2})/2$ nodes are eliminated: At parallel-operation at least E/2 edges are split into at most $2E/(N^{1/2})$ subgraphs, so at least one of the subgraphs contains $(N^{1/2})/4$ edges, which are incident with $(N^{1/2})/2$ nodes.

Let F(N,E) be the time required to find a maximal matching using algorithm 10.2

$$F(N,E) < \max \begin{cases} F(N\ ,\ E/2) + C_1\ N^{1/2}\ \log_2 d \\ F\{N-(N^{1/2})/2\ ,\ E-(N^{1/2})/4\} + C_2\ \log_2 E\ \log_2 d \end{cases}$$

One can perform sequential operation at most $\log_2 E$ times before eliminating all the edges, which requires $O(N^{1/2}\cdot\log_2 d\cdot\log_2 E)$ time altogether.

Claim

Parallel operation requires at most $C_2C_3N^{1/2}E\log_2 d$ time before eliminating all the edges.

Proof

Assume that for each N'<N

$$F(N',E) < C_2C_3\cdot N^{1/2}\cdot\log_2 E\cdot\log_2 d$$

Then for N we get

$$F(N,E) < F(N-(N^{1/2})/2\ ,\ E-(N^{1/2})/4) + C_2\log_2 E\ \log_2 d \leq$$

(by the induction hypothesis)

$$C_2\cdot C_3\cdot(N-((N^{1/2})/2))^{1/2}\cdot\log_2 E\cdot\log_2 d + C_2\cdot\log_2 E\cdot\log_2 d \leq$$

$$C_2\cdot\log_2 E\cdot\log_2 d\cdot(C_3N^{1/2}(1 - (N^{-1/2})/2)^{1/2} + 1) <$$

$C_2 \cdot \log_2 E \cdot \log_2 d \cdot (C_3 \cdot N^{1/2}(1 - (N^{-1/2}/4)) + 1) <$

$C_2 \cdot \log_2 E \cdot \log_2 d \cdot C_3 \cdot N^{1/2}$

if $C_3/4 > 1$

and $F(N,E) < C_2 \cdot C_3 \cdot \log_2 E \cdot \log_2 d \cdot N^{1/2}$

And the claim is proved.

Therefore the algorithm terminates in at most

$O(N^{1/2} \log_2 E \log_2 d)$ time.

10.5 Third maximal matching algorithm for bipartite graphs

Like the previous algorithm, this algorithm does not assume knowledge of the groups of the bipartite graph, and it uses complete binary splitting as a subroutine.

At each step of the algorithm, let the subgraph containing the still unmatched nodes and the edges incident with them have degree d, and let $k=\lceil \log_2 d \rceil$. Consider the nodes with degree at least 2^{k-1}. At each call to complete-binary-splitting the graph is split into 2^k subgraphs. At least one of the subgraphs found by the splitting contains half of those nodes. The edges in such a subgraph are added to the matching, and the number of nodes with degree at least 2^{k-1} is reduced to half its previous value. After at most $\log_2 N$ iterations, there are no more nodes with such a high degree and k is reduced at least by 1. In the algorithm below we added a redundant variable in order to discuss the number of iterations of the algorithm.

Let G(A,B) have maximal degree d.

Algorithm 10.3

!each proc par do

initiation

-Set all the nodes and edges of G to be active.

G' denotes the subgraph of active nodes and edges.

$k \leftarrow \lceil \log_2 d \rceil$; $i \leftarrow 0$

end intiation

procedure 1

1) -Find the degree d(v) of each node in G'.

-Find d the degree of G'.

if $k < \lceil \log_2 d \rceil$

then $k \leftarrow \lceil \log_2 d \rceil$ (k is reduced)

$i \leftarrow 0$

else $i \leftarrow i+1$.

-Tag all the nodes whose degree is greater than 2^{k-1}.

-if G' is empty terminate, a maximal matching was found

2) -call complete binary splitting for G'.

-In each subgraph created by the splitting tag each edge which is incident with a tagged node.

3) -Find the subgraph (in the splitting) with a maximum number of tagged edges.

-Add the edges in this subgraph to the matching.

-Delete from G' all the edges just added to the matching, all the nodes incident with them, and all the edges adjacent to these.

call procedure 1

end of procedure 1

end par do

end of algorithm

Theorem 10.4

Algorithm 10.5 finds a maximal matching in a bipartite graph in at most $O(\log^2 d \cdot \log N \cdot \log E)$ time.

Proof

Each recursive call (or iteration) of the algorithm calls complete binary splitting which requires $O(\log_2 E \cdot \log_2 d)$ time.

For a given value of k, at each call to the procedure at least half the nodes whose degree is in the range 2^{k-1} to 2^k are matched and eliminated from G'. For a given value of k, i can be increased at most $\log_2 N$ times before k is reduced by 1, k can be reduced at most $\log_2 d$ times before it is reduced to 0, and the algorithm terminates.

Therefore the total time does not exceed

$$O(\log_2^2 d \cdot \log_2 N \cdot \log_2 E) \text{ time.}$$

Combining the three algorithms, gives the results claimed in the introduction to the chapter.

REFERENCES

AHO A.V., HOPCROFT J.E., ULLMAN J.D. [1974] The design and anlysis of computer algorithms. Addison Wesley, Reding,Mass.

ANGLUIN D.,VALIANT L.G. [1979] Fast probabilistic algorithms for Hamiltonian circuits and matchings. J. of Computers and System Sci., 18:2, pp. 155-193.

ANDERSEN S. [1977] The looping algorithm extended to base 2^t rearrangeable switching networks. IEEE Trans. on communications COM25 pp. 1057-1063.

ARJORANDI E., CORNEIL D.G. [1975] Parallel computations in graph theory. Proc. 16th symp. on Foundations of Computer Science, pp. 13-18.

BATCHER K.E. [1968] Sorting networks and their applications. Proc. AFIPS conf. 1968, SJCC Vol 32, AFIPS Press,Montvale N.J. ,pp. 307-314.

BEIZER D. [1962] The analysis of signal switching networks. Proc. Symp. on Mathematical Theory of Automata, Brooklyn Polytechnic Institute, Brooklyn, New-York, pp.563-576.

BENEŠ V.E. [1965] Mathematical Theory of Connecting Networks and Telephone Traffic. Academic Press, New York.

BERGE C. [1973] Graphs and Hypergraphs. North-Holland, Amsterdam.

BORODIN A. [1977] On relating time and space to size and depth. Siam J. Comp. 6, pp. 733-743.

CANTOR D.G. [1971] On non blocking switching networks. Networks 1, John Wiley and Sons, pp 367-377.

DIEUDONNÉ [1970] Une propriete des racines de l'unite. Revista de la Union Mathematica Argentina 25, pp.1-3.

FLYNN M. [1966] Very high speed computing systems. Proc. IEEE 54, pp. 1901-1909.

FORTUNE S., WYLLIE Y. [1978] Parallelism in random access machines. Proc. 10-th ACM Symp. on Theory of Computing, pp.114-118.

GABBER O., GALIL Z. [1979] Explicit construction of linear size superconcentrators. Proc. 20-th IEEE Symp. on Foundations of Computer Science, pp. 364-370.

GABOW H.N. [1976] Using Euler partitions to edge colour bipartite multigraphs. International Journal of Computer and Information Sciences 5, pp. 345-355.

GABOW H.N., KARIV O. [1978] Algorithms for edge colouring bipartite graphs. Proc. 10-th ACM Symp. on Theory of Computing, pp. 184-192.

GABOW H.N., KARIV O. [1980] Algorithms for edge colouring bipartite graphs and multigraphs. To appear.

GAVRIL F. [1975] Merging with parallel processors. CACM 18, pp. 588-591.

GOKE L.R., LIPOVSKY G.J. [1973] Banyan networks for partitioning multiprocessor systems. PROC 1st Conf. Comput. Architecture, pp. 21-28.

GOLDSCHLAGER L.M. [1978] A unified approach to models of

synchronous parallel machines. Proc. 10-th ACM Symp. on Theory of Computing, pp. 89-94.

GUIBAS L.Y., KUNG H.T., THOMPSON C.D. [1979] Direct VLSI implementation of combinatorial algorithms. Proc. Conf. on VLSI: Architecture, Design, Fabrication , Caltech.

HIRSCHBERG D.S. [1976] Parallel algorithms for the transitive closure and the connected components problems. Proc. 8th ACM Symp. on Theory of Computing, pp.55-57.

HIRSCHBERG D.S. [1978] Fast parallel sorting algorithms. CACM 21, pp. 657-661.

HOPCROFT J.E., KARP R.M. [1973] An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. COMPUT 2,4 pp.225-231

JOEL J.A.E. [1968] On permutation switching networks. Bell sys. Tech. J. 47, pp 813-822.

KANT R.M., KIMURA T. [1978] Decentralized parallel algorithm for matrix computation. ACM SIGARC. 6, Conf. Proc. 5th Symp. on comp. architecture pp. 96-100.

KNUTH D.E. [1968] The Art of Computer Programming. Vol. 1: Fundamental algorithms. Addison Wesley, Reading Mass.

KNUTH D.E. [1973] The Art of Computer Programming. Vol. 3: Sorting and Searching. Addison Wesley, Reading Mass.

KUCK D.J. [1978] The Structure of Computers and Computations, Vol. 1. John Wiley& Sons, New York.

KUNG H.T. [1980] The stucture of parallel algorithms. To appear in Advances in Computers 19, Academic Press, New York.

LADNER R.E.,FISCHER M.J. [1977] Parallel prefix computation. ICPP, pp.218-223.

LAWRIE D. [1975] Access and alignment of data in an array processor. IEEE Trans. on Computers C24, pp. 1145-1155.

LEV G., PIPPENGER N.J., VALIANT L.G. [1980] Fast parallel algorithm for routing in permutation networks. To appear in IEEE. Trans. on Computers.

MASSON G.M. [1977] Binomial switching networks for concentration and distribution. IEEE Trans. on Communications COM25, pp. 873-883.

MASSON G.M., GINGHER G.C., NAKAMURA S. [1979] A sampler of circuit switching networks. Computer, June 1979, pp.32-48.

MARGULIS G.A. [1973] Explicit construction of concentrators. Problemy Peredachi Informatsii 9, pp. 71-80. English translation in Problems of Information Transmition, Plenum Publishing Corp., New York, 1975.

MORGENSTERN J. [1973] Note on a lower bound of the linear complexity of the Fast Fourier Transform. J. ACM 20, pp. 305-306.

MUNRO I., PATERSON M. [1973] Optimal algorithms for parallel polynomial evaluation. J. Comp. System Science 7, PP.189-198.

OFMAN J.P. [1965] A universal automaton. Trans. of the Moscow Mathematical Society 14. English translation by American Mathematical Society, Providence R.I., 1967, pp.200-215.

ORE [1962] Theory of Graphs. American Mathematical Society Colloquium Publications Vol.38.

PATERSON M.S. [1976] An introduction to Boolean function complexity. Asterique 38-39, pp. 183-201.

PINSKER M. [1973] On the complexity of a concentrator. Proc. of the 75th International Teletraphic Conf. Stockholm. pp.318/1-318/4.

PIPPENGER N., VALIANT L.G. [1976] Shifting networks and their applications. J.ACM. 23, pp. 423-432.

PIPPENGER N. [1977a] Superconcentrators. SIAM J. Comp. 6, pp. 298-304.

PIPPENGER N. [1977b] Generalized connectors. IBM Res. Rep. RC-6532, T.J. Watson Research Center, Yorktown Heights N.Y.

PIPPENGER N. [1978a] Complexity Theory. Scientific American 238, pp.90-100.

PIPPENGER N. [1978b] On rearrangeable and nonblocking switching networks. J. Comp. System Science 17, pp.145-162.

PIPPENGER N. [1979] A new lower bound for the number of switches in rearrangeable networks. IBM. Res. Rep. RC 7890 (34215)

PRATT V.R., STOCKMEYER L.J. [1976] A characterization of the power of vector machines. J. Comp. System Science 12, pp. 198-221.

PREPARATA F.P. [1978] New parallel sorting schemes. IEEE Trans. on Computers C27, pp.669-673.

PREPARATA F.P., VUILLEMIN J. [1979] The cube connected cycles: a versatile network for parallel computations. Proc. 20-th IEEE Symp. on Foundations of Computer Science, pp.140-147.

SAVAGE J.E. [1976] The complexity of computing. John Wiley and Sons, New-York.

SCHNORR C.P. [1974] Zwei lineare untere Schranken fuer die Komplexitaet Boolescher Funktionen. Computing 13, pp. 155-171.

SIEGEL H.J. [1977] The universality of various types of SIMD machine interconnection networks. ACM SIGARCH 7, pp.70-79.

SIEGEL H.J. [1979] Interconnection networks for SIMD machines. Computer , June 1979, pp. 57-65.

STONE H.S. [1971] Parallel processing with the perfect shuffle. IEEE Trans. on Computers C20, pp.153-161.

THOMPSON C.D., KUNG H.T. [1976] Sorting on a mesh connected computer. Proc. ACM-SIGACT Symp. on Theory of Computing, pp. 58-64.

THOMPSON C.D. [1978] Generalized connection networks for parallel processor intercommunication. IEEE Trans. on Computers C27, pp.1119-1125.

TOMPA M. [1978] Time space tradeoffs for computing functions using connectivity properties of their circuits. Proc. 10-th ACM Symp. on theory of computing, pp.196-204.

VALIANT L.G. [1975a] On non linear lower bounds in computational complexity. Proc. 7-th ACM Symp. on theory of computing, pp. 45-53.

VALIANT L.G. [1975b] Parallelism in comparison problems. SIAM J. Comput. 4, pp. 349-355.

VALIANT L.G. [1977] Graph-theoretic arguments in low level complexity, Proc. 6-th Symp. on Mathematical Foundations of

Computer Science, Lecture Notes in Computer Science no. 53, Springer, pp. 162-176.

WAKSMAN A. [1968] A permutation network. J. ACM 15, pp.159-163.

WINOGRAD S. [1978] On computiong the Discrete Fourier Transform. Math. of Computation Vol 32, No 141, pp.175-199.